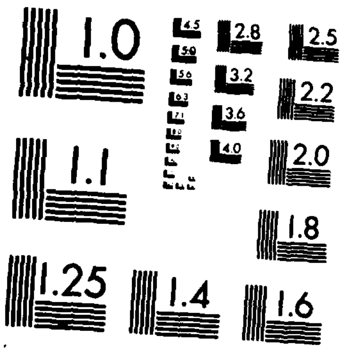
AD-A173 599 LONG-RANGE TECHNOLOGICAL IMPACT ON COMPUTER-AIDED 1/3
PRODUCT DEVELOPMENT AT (U) WASHINGTON UNIV ST LOUIS MO
DEPT OF COMPUTER SCIENCE G ROMAN ET AL JUL 86
UNCLASSIFIED RADC-TR-86-49 F30602-83-K-0065 F/G 9/2 NL


1.0
4.5
5.0
5.6
6.3
7.1
8.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

(12)

RADC-TR-86-49
Final Technical Report
July 1986

LONG-RANGE TECHNOLOGICAL IMPACT ON COMPUTER-AIDED PRODUCT DEVELOPMENT AT DMA

Washington University

Gruia-Catalin Roman, R. Howard Lykins, Robert K. Israel,
H. Conrad Cunningham and Michael E. Ehlers

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

This effort was funded totally by the Defense Mapping Agency

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NITS it will be releasable to the general public, including foreign nations.

RADC-TR-86-49 has been reviewed and is approved for publication.

APPROVED: Roger B Panara

ROGER B. PANARA
Project Engineer

APPROVED: Raymond P. Urtz Jr

RAYMOND P. URTZ, JR.
Technical Director
Command and Control Division

FOR THE COMMANDER: Richard W. Pouliot

RICHARD W. POULIOT
Plans & Programs Division

DESTRUCTION NOTICE - For classified documents, follow the procedures in DOD 5200.22-M, Industrial Security Manual, Section II-19 or DOD 5200.1-R, Information Security Program Regulation, Chapter IX. For unclassified, limited documents, destroy by any method that will prevent disclosure of contents or reconstruction of the document.

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COEE) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

ADA 173599

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

1b. RESTRICTIVE MARKINGS
N/A

2a. SECURITY CLASSIFICATION AUTHORITY
N/A

2b. DECLASSIFICATION / DOWNGRADING SCHEDULE
N/A

3. DISTRIBUTION / AVAILABILITY OF REPORT
Approved for public release;
distribution unlimited

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
N/A

5. MONITORING ORGANIZATION REPORT NUMBER(S)
RADC-TR-86-49

6a. NAME OF PERFORMING ORGANIZATION
Washington University

6b. OFFICE SYMBOL *(If applicable)*

7a. NAME OF MONITORING ORGANIZATION
Rome Air Development Center (COEE)

6c. ADDRESS *(City, State, and ZIP Code)*
Department of Computer Science
Campus Box 1045
St Louis MO 63130

7b. ADDRESS *(City, State, and ZIP Code)*
Griffiss AFB NY 13441-5700

8a. NAME OF FUNDING / SPONSORING ORGANIZATION
Defense Mapping Agency

8b. OFFICE SYMBOL *(If applicable)*
STT

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
F30602-83-K-0065

8c. ADDRESS *(City, State, and ZIP Code)*
Building 56, U.S. Naval Observatory
Wash DC 20305

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.
63701B	3205	TG	50

11. TITLE *(Include Security Classification)*
LONG-RANGE TECHNOLOGICAL IMPACT ON COMPUTER-AIDED PRODUCT DEVELOPMENT AT DMA

12. PERSONAL AUTHOR(S)
Gruia-Catalon Roman, R. Howard Lykins, Robert K. Israel, H. Conrad Cunningham, Michael E. Ehlers

13a. TYPE OF REPORT
Final

13b. TIME COVERED
FROM Jan 83 TO Jan 86

14. DATE OF REPORT *(Year, Month, Day)*
July 1986

15. PAGE COUNT
286

16. SUPPLEMENTARY NOTATION
This effort was funded totally by the Defense Mapping Agency

17. COSATI CODES

FIELD	GROUP	SUB-GROUP
09	02	

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Formal Design Methodology
Technology Evaluation
Software Development Environments
Geographic Data Processing
Requirement Specification and Validation

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

Increasing demands are being placed on DMA by DOD organizations for Mapping, Charting, and Geodesy products. To meet these demands, DMA must automate labor intensive tasks, particularly with regards to photo interpretation and software development and maintenance practices must be responsive to changes in DMA needs. Geographic Data Processing (GDP) systems are the key vehicle for automation of the production process. The DMA Modern Programming Environment (MPE) implementation effort offers the means for an evolutionary upgrade for changing software programming needs. Technology advances in areas such as artificial intelligence, database modelling, high speed processing, problem oriented languages, Ada, etc. are fertile for exploitation. However, improper application of the technologies could prove to be costly as well as of little value.

This study has produced a requirements specification and validation methodology which could significantly reduce the risk associated with building GDP systems through the development
(Cont'd on reverse)

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT
□ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT □ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL
Roger B. Panara

22b. TELEPHONE *(Include Area Code)*
(315) 330-4063

22c. OFFICE SYMBOL
RADC (COEE)

83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Block 19. Abstract (Cont'd)

of inexpensive rapid prototypes. The strategy is to build simulations for the functionality of the proposed system. By interactive play-back of the simulation script, functional adequacy and potential impact on production can be assessed prior to development or purchase. The methodology also allows one to focus the scope of technology studies on particular production problems and specific design solutions.

The relationship between the DMA MPE implementation and new GDP system was examined as part of the production upgrade effort at DMA. The study concluded that the MPE can smoothly evolve and provide adequate support to changing needs. Clearly indentified trends and not speculative ideas of the future development of computer and software technologies were used in arriving at this result.

TABLE OF CONTENTS

1. EXECUTIVE SUMMARY

1.1. GOALS

The Defense Mapping Agency (DMA) production plan is determined by the Mapping, Charting, and Geodesy (MC&G) needs of the many Department of Defense (DoD) organizations. Because the generation of most DMA products involves some form of computer processing of geographic data, DMA's overall productivity and responsiveness depends, to a large extent, on the productivity and quality of the software employed in support of the MC&G production.

In recognition of this fact, DMA has established several major programs aimed at improving the productivity and quality of its data processing activities. These include the Software Improvement Program, the development of several new major Geographic Data Processing (GDP) systems (e.g., Universal Rectifier, Digital Stereo Compiler, Clustered Carto System, etc.) and the Modern Programming Environment (MPE). They are part of a general thrust toward achieving a more effective computer-aided product development process and responsiveness to evolving needs.

The fact that the needs of DMA will change, and at a rapid rate, is obvious from the almost exponential growth in sophisticated military hardware. What is not clear, however, is the degree to which the technology that fosters this growth can also be used to enhance DMA productivity. A major goal of this research effort is to shed light on this issue and in so doing, facilitate the timely transfer of new technology into the DMA production environment. We approach this issue from three directions:

(1) *Technological Opportunities*—This involves the identification of technological elements and trends that hold the potential for significant advances in computer processing of geographic data at DMA.

(2) *Technology Transfer*—This is concerned with ways by which DMA may exploit the technology considered above in order to achieve higher levels of productivity and responsiveness to increasing demands for MC&G products. We see GDP systems as becoming an important vehicle for accomplishing the transfer of new technology into DMA production.

(3) *Software Practices Dynamics*—This examines the changes in the DMA software practices that may take place as a result of the emergence of GDP systems and the related impact on the requirements for the far-term MPE Facility.

1.2. MOTIVATION

The findings reported here will benefit DMA in two related ways. First, our investigation into technological trends and opportunities useful in the development of powerful new GDP systems assists planning and design efforts in an area critical to the enhancement of the DMA's production capabilities. Second, we provide the designers of the far-term MPE Facility much needed data regarding the manner in which software practices at DMA will be affected by the growing presence of GDP systems.

Near-term plans recognize the need for the MPE Facility to be adaptive and to be capable of assimilating new tools in response to changes in the DMA needs (the mere introduction of the facility will trigger new expectations). However, planning for the evolution of the MPE appears to be based on the implicit assumption that the software development practices at DMA will not change in any fundamental way. This work is motivated by the desire to examine the degree to which this assumption is valid and to assess the impact of our findings on the second phase of the MPE program which is scheduled to start after 1985. It is our conclusion that a strong commitment to the use of GDP systems will alter the character of the MPE Facility in significant ways without eliminating the need for such a facility.

In order to reach this conclusion we reviewed the current DMA efforts to build GDP systems, studied the role they are anticipated to play in the DMA production process, and investigated methodological aspects related to their development. Based on this information we established the far-term impact of GDP systems on the MPE Facility, and identified ways to guarantee proper cooperation between the MPE Facility and the GDP systems.

1.3. TASKS

The tasks we performed under the contract are:

(1) Identification of technological opportunities that could be exploited (through the use of GDP systems) for the benefit of DMA production.

(2) Study of the GDP systems requirements and design methodologies in order to establish the role GDP systems will play at DMA and the demands they will place on the far-term MPE Facility.

(3) Consolidation of the far-term MPE requirements and development of a far-term MPE Facility operational model that is useful to the designers of the far-term MPE Facility and achievable (in an evolutionary manner) from the near-term MPE Facility configuration.

The remaining sections of the executive summary provide an overview of the activities carried out under each task.

1.4. GENERAL TECHNOLOGY ASSESSMENT

For part of this task, we reviewed a number of GDP systems in order to gain a thorough understanding of their nature. Based on this review and on a general knowledge of technological trends, a composite picture of the requirements for a general GDP system has been created and the areas of technology which are relevant to the design and implementation of a GDP system have been identified.

Because we consider GDP systems from the standpoint of their use as a production tool, the focus is on technology which will improve the overall productivity of the system with respect to:

- throughput in product development and generation;
- accuracy, completeness, and quality of products;
- responsiveness to changes in the nature and volume of production requirements;
- low maintenance and retooling costs;
- robustness and survivability; and
- security.

When possible, currently available technology which meets the GDP requirements is identified. Specific areas of research are identified, however, for those areas where current technology is either insufficient or continued technological development is necessary for the successful long-term operation of a GDP system.

Among the various technological factors being considered in this report, the Ada® language receives special attention. The opportunities and the challenges offered by Ada are reviewed. This leads to a discussion of the Ada training needs at DMA. A sample training program is illustrated in the section on the development of an Ada culture within the organization. Finally, we put forth a proposal for a low risk strategy leading to the integration of Ada into the DMA production environment.

1.5. SPECIFICATION AND VALIDATION OF GDP REQUIREMENTS

We see GDP systems as the key vehicle for automating significant parts of the DMA production process and thus reducing the high level of human labor currently required (particularly with regard to photo interpretation). Due to the extreme requirements placed on GDP systems needed by DMA (size, data volume, production throughput, etc.) and due to limitations in the current state of the art, these systems are expensive and difficult to design.

As part of this task we propose a requirements specification and validation methodology. This methodology could reduce significantly the risk associated with building GDP systems for DMA through the development of inexpensive rapid prototypes. These prototypes can be subjected to technical and human factors evaluations involving both technica' experts and production personnel.

Rooted in the notion of rapid prototyping, our strategy is to build simulations for the functionality of some proposed system, or part thereof, and to record intermediary results of the simulation for the purpose of an interactive real-time play-back of the simulation script under user control. Built-in adjustable delays reproduce the performance characteristics of the technical solution under consideration. In this way both the functional adequacy and the potential production impact may be clearly identified prior to committing to the actual purchase or development of the system. The evaluators may be either technical experts or actual production personnel and the simulations may be used both as a means for requirements validation and for technical evaluation of proposed designs as well as an avenue for building and evaluating training materials for new production systems.

Another important component of this task is the establishment of a formal foundation for the specification of GDP requirements. The emphasis is placed on modelling data and knowledge requirements rather than processing needs. A subset of first order logic is proposed as the principal means for constructing formalizations of the GDP requirements in a manner that is independent of the data representation. Requirements executability is achieved by selecting a subset of logic compatible with the inference mechanisms available in the programming language Prolog. Significant GDP concepts such as time, space accuracy and security classification have been added to the formalization without losing Prolog implementability or separation of concerns. Rules of reasoning about time, space, accuracy (based on positional, temporal and fuzzy logic) and security classification may be compactly stated in a subset of second order predicate calculus and may be easily modified to meet the particular needs of a specific application. Multiple views of the data and knowledge may coexist in the same formalization. The feasibility of the approach has been established with the aid of a tentative Prolog implementation of the formalism. The implementation also provides the means for graphical rendering of logical information on a high resolution color display.

1.6. TECHNOLOGY EVALUATION METHODS

The relevance of our requirements specification and validation methodology is captured by two important attributes. First, it makes explicit the impact of technology on the production environment. Second, it allows one to focus the scope of technology studies on the needs of a particular class of production problems and on a specific domain of design solutions. This task explores some of the technical details involved in exploiting the latter aspect of our methodology.

The result is the formulation of an integrated approach to design and technology evaluation. Fundamental to our approach is the reliance on system level models that enable the designer to formulate and answer questions regarding the system's logical and performance characteristics when the interaction between the hardware and the software is important, i.e., when the impact of faults, failures, communication delay, hardware selection, scheduling policies, etc. must be considered. In the simplest terms, our concern extends beyond the traditional (isolated) software and hardware concerns by addressing the issue of software correctness and performance characteristics when running on a particular distributed hardware architecture and using a particular operating system.

The technical foundation for the methodology relies on the concept of *Virtual System*. Virtual Systems represent either abstractions of existing systems or definitions of proposed systems. A virtual system consists of six components. The *Functionality* is an abstraction of the processes which carry out the system function (e.g. the applications software). The *Architecture* captures the overall hardware organization and distribution of the system. The *Scheduler* together with the *Allocation* define the relationship between functionality and architecture, and the changes which the relationship undergoes with time. *Allocation,* while related to the notion of partition present elsewhere, is much more general. It covers static and dynamic allocation of functions (in the functionality) to processors (in the architecture) and the allocation of time and space on an individual processor to the functions associated with it. The *Performance Specification* is an abstraction of both workload characteristics (the environment model is an integral part of the virtual system) and measurement probes. The performance specification may be used to explicitly state the assumptions made by designers regarding the characteristics of the environment and of the system components to be utilized in the realization of the system. The performance specification is coordinated with the rest of the model via the *Instrumentation.*

We experimented with specifying virtual systems using a language called CSPS *(Communicating Sequential Processes with Synchronization).* Several system models were built. They share the same functionality but alternate supporting architectures: uniprocessor, dual-processors with point to point communication, dual-processors with bus communication and dual-processors with point to point communication and dynamic reallocation.

1.7. FAR-TERM MPE OPERATIONAL CONCEPT

This task addresses the definition of an operational concept for the Defense Mapping Agency's (DMA) far-term Modern Programming Environment (MPE). A *modern programming environment* is a coordinated collection of computers and software tools dedicated to supporting the development and maintenance of computer software. The goals of an MPE are two-fold: to increase the productivity of the software project personnel (analysts, designers, programmers, managers, etc.) and to improve the quality (reliability, correctness, maintainability, etc.) of the software that they develop and maintain. The MPE is "modern" in the sense that it seeks to integrate "state-of-the-art" software and computer technologies into a coherent facility. Moreover, to remain modern, the MPE must be flexible enough to evolve gracefully as better technologies emerge. DMA contracted with the Data Systems Division of General Dynamics Corporation to develop the DMA MPE and introduce it into DMA operations. Our contract addresses how this initial facility should evolve.

The objective of this task is to study the potential impacts of the changing MPE requirements in the context of currently identifiable technological trends. In this study we examine five technological areas:

- tools,
- human interfaces,
- methodologies,
- architectures,
- software development environments.

To make our study realistic, we concentrate on clearly identifiable trends and avoid purely speculative ideas on the future development of computer and software technologies.

In approaching this task, we assume a functionalist perspective. First, we identify the evolutionary processes to which the MPE Facility will be subjected because of the changing needs of DMA. With this view of the changing requirements, we then derive an understanding of th structures and procedures that will most likely foster facility evolution. Using this understanding of the requirements and the technological trends, we then suggest an evolutionary path for the MPE Facility.

1.8. CONCLUSION

The key concern of our study was the relation between the proposed MPE Facility and the new generation of GDP systems being planned and built as part of a general effort to upgrade the production capabilities of DMA. Our study shows that the MPE Facility is in a very good position to provide adequate support both for in-house software development and maintenance and for the maintenance and enhancement of the GDP systems software. While the latter will require the assimilation of additional tools and methodologies, the MPE Facility has the capability to respond smoothly to new needs and to grow.

A second area of concern was the means by which the risks associated with the development of sophisticated GDP systems may be reduced. The answer rests with the development of new methodologies and tools (for requirements validation, system design and technology evaluation) which are capable of maintaining the relevance of the production objectives throughout the system life-cycle and of making evident the impact of design and technological choices on the production process the GDP system is meant to support. We have shown that the development of such methodologies is feasible and have identified a technical foundation on which they may be built.

2. GENERAL TECHNOLOGY ASSESSMENT

This part of the report provides a tutorial on the nature of GDP systems (Section 2.1) and the technological areas on which they rely (Section 2.2). Pressing research needs in the GDP area (Section 2.3) and the potential near-term impact of Ada® (Section 2.4) are also discussed. The reader not interested in the issues addressed here is advised to read Section 2.1 before skipping to Sections 3, 4, or 5.

2.1. GDP REQUIREMENTS DEFINITION

2.1.1. Introduction

We have reviewed a number of Geographic Data Processing (GDP) systems in order to gain a thorough understanding of their nature. Based on this review and on a general knowledge of technological trends, a composite picture of the requirements for a general GDP system has been created. The purpose of this section is to present this composite model and to provide the context for later discussions of these systems.

In general, a GDP system performs the following functions. First, it receives information about the physical world in a large variety of forms, from satellite sensor data to written survey reports. It must then organize this information and maintain it in a form which is suitable for both internal use and product generation. The GDP system must also ensure the consistency, completeness, quality, and security of the information it maintains. Finally, the GDP system generates a varied set of products which can range from contour maps to geographic databases for use by other systems.

A model of such a GDP system is illustrated in Figure 2.1-1. In this figure, the names of objects are printed in capital letters, and the names of actions that are performed on these objects are printed in lower-case letters. The first action which is performed by the GDP system is data acquisition, in which information about the physical world (DATA SOURCES) is encoded in a form that is processable by the system directly (INPUT DATA). Once the data has been acquired, a number of processing steps (transformations) may be performed on the data in order to alter its format, remove or correct errors, and ensure its compatibility with information already in the GDP system. When the transformations are completed, the input data is analyzed and incorporated into a general geographic information database.

In addition to FACTUAL DATA, the geographic information database incorporates WORLD KNOWLEDGE captured by models of the structure of the physical world in order to provide a powerful means of deriving new information from the basic facts stored in the system. This provides the ability to evaluate the semantic integrity of information, determine the security classification of existing and derived information, and reinterpret information based on new models which can be continuously developed and added to the system.

In order to generate products, the information required for each product must be synthesized from the geographic information database and put in a form which is

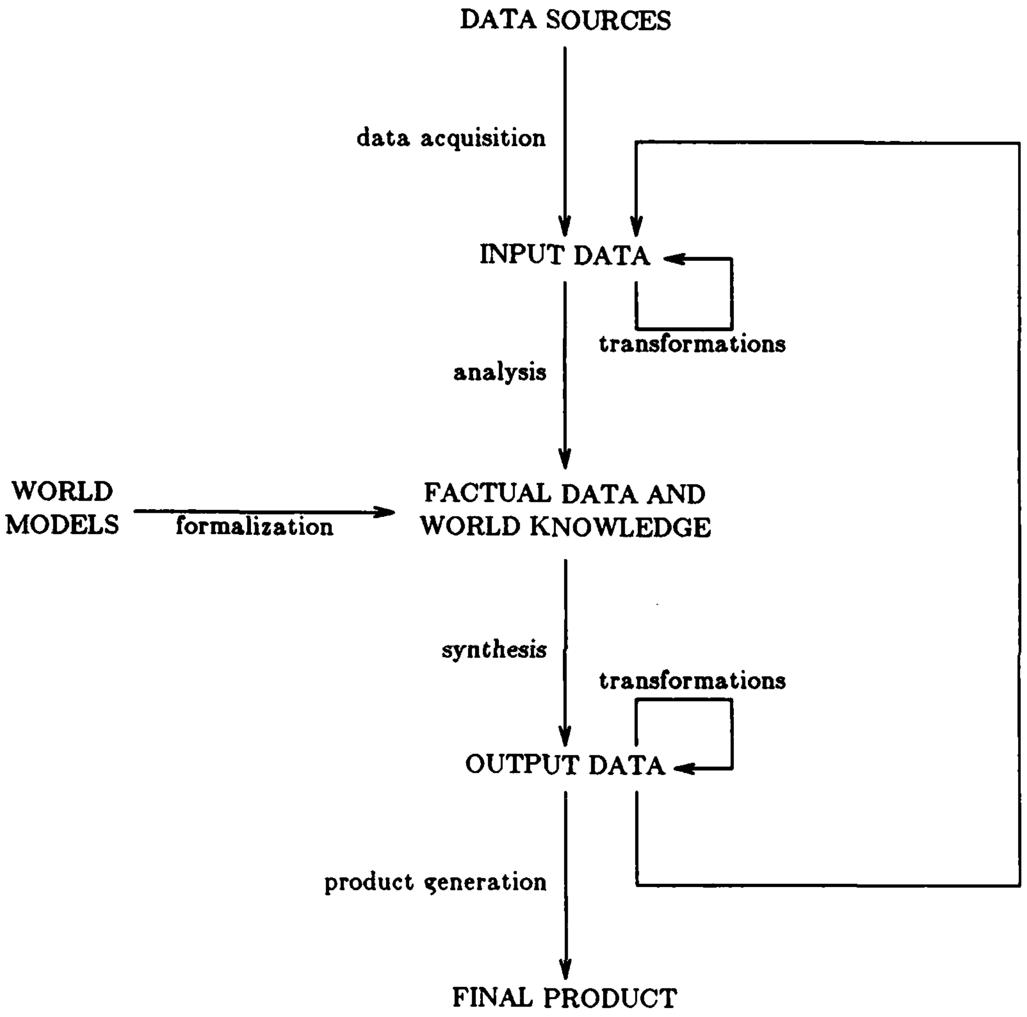


Figure 2.1-1. GDP System Requirements Model

amenable for processing (OUTPUT DATA). This data may undergo a number of transformations such as format changes or information extraction before it can be used for product generation. The generation of FINAL PRODUCTS such as maps, reports, and databases is the last step performed by the GDP system.

The following subsections discuss the requirements for a GDP system based on the general model presented above. Section 2.1.2 deals with the requirements for data acquisition; Section 2.1.3 covers input data representation, analysis, and transformation; Section 2.1.4 discusses the concepts of factual data, world knowledge, and their management; Section 2.1.5 deals with output data representation, synthesis, and transformation; and Section 2.1.6 covers product generation. This characterization of GDP system requirements is concluded in Section 2.1.7 with a discussion of the major non-functional constraints which must be met by these systems.

2.1.2. Data Acquisition

Data acquisition is the process by which information about the physical world is put into a form which is processable by the GDP system. The information sources that are typically used in GDP systems include (but are not limited to):

- cartographic maps,
- mono photographs,
- stereo photograph pairs,
- remotely sensed data,
- survey records, and
- written reports.

A variety of methods have been developed for converting information from the above sources to a machine processable form. In the line digitization process, existing maps or photographs are placed on a digitizing tablet and the features of interest (points, elevation contours, rivers, roads, political boundaries, etc.) are traced and recorded for later processing. In stereophotogrammetric processing, a stereo pair of photographs are automatically analyzed to produce a set of elevation points. In some cases, images of varying quality may be digitized (encoded as a grid of pixels) and stored. Digital or analog electronic data recorded from remote sensing systems may require some form of conversion from their raw form to a form that is processable by the GDP system hardware and software. Finally, textual information such as names, survey reports or other field reports may need to be summarized and input manually.

Two additional aspects of the data acquisition process that must be handled by the GDP system are accuracy determination and data security. As part of the collection process, the system must assess the accuracy and completeness of each data item acquired. The management of this information is important for maintaining the consistency of system data during later processing steps and for improving the quality of products produced by the system. In addition to accuracy, the security classification of the data must also be determined (based on the type of data, its source, accuracy, region of coverage, etc.). Once the data is entered into the GDP system, it is the system's responsibility to ensure that the security of the data is not compromised. Some aspects of security, such as access to raw data items (i.e. photographs), are not within the scope of this work.

2.1.3. Input Data Representation, Analysis, and Transformation

The data representations used to store input data vary greatly depending on the input source, the type of information, and the processing steps which need to be performed on the data. Common data representations include raster digital images, vector sets derived from line tracing on a digitizing tablet, elevation matrices such as those derived from stereophotogrammetric processing, and other surface variable matrices which may be derived from remotely sensed sources or survey data (examples of such would be matrices depicting land use, terrain type, or population density).

One of the primary difficulties with input data is the presence of anomalies and discrepancies in the data. These errors in the data can be caused by noise and interference in the collection process, malfunctions in the collection equipment, problems in the models on which the collection process is based, or human error. Other problems which need to resolved include gaps in the data, inconsistencies with respect to existing data, and the presence of extraneous information which must be filtered out.

Transformations on the data fall roughly into three categories: changes in data representation (such as between surface contours and elevation matrices), correction of anomalies in the collection process, and corrections which are applied to the data in order to preserve consistency with existing information. Changes in the data representation are commonly done in order to allow for efficient error detection, error correction, or abstraction of information. As an example, data entered as a set of contour lines may need to be changed to elevation matrix form in order to determine its consistency with existing data already in that form. A single piece of information may take on many forms during the course of input processing. The correction of anomalies in the input process generally depends on the source of the data collection. Typical correction procedures include correction of skew and non-linearities in remotely sensed data, transformation of coordinates into some internal standard, interpolation of data from a non-uniform sample grid to a uniform sample matrix, and various image enhancement techniques. Operations to maintain the consistency of input data with existing data all require the existing data to be abstracted from the GDP database and formatted appropriately for checking against the input data. Common operations along these lines include panelling, feathering, and hole filling.

The final step in the input data processing is the analysis of the data to obtain the factual information maintained by the GDP system. Operations in this step include reformatting of the data into a form compatible with the GDP factual information store, extraction of additional information from the data (such as feature analysis), various checks for consistency of the new and existing data, and the correction of existing information based on the new data.

2.1.4. Factual Data, World Knowledge, and Their Management

The factual data maintained by the GDP system can be viewed as a collection of objects which possess both geometric and non-geometric attributes. Objects usually represent entities such as rivers, cities, or countries. The types of geometric attributes which may be specified include shape (point, line, area, volume), position, line variables (surface profiles, river elevation), surface variables (land elevation, terrain type), and volume variables (ocean temperature, geologic information). Non-geometric attributes include names, other textual annotations, quality or accuracy assessment, and security classification.

Each one of the attributes listed above may be qualified temporally or probabilistically. Temporal qualifications include such things as the time period over which the attribute is valid or the change of the attribute with time. Time may be considered to be either absolute (e.g. object o exists at time t), or cyclic (e.g. river r is dry during the months July through September). Probabilistic qualifiers assess the certainty with which a particular attribute is valid, thus allowing for information whose truth is uncertain to be analyzed by the system (e.g. a missile is present in silo s with probability p). In current systems these attributes are handled in an ad hoc manner, if at all. Current trends indicate, however, that temporal and probabilistic concepts will play an important role in future systems.

In addition to providing a means of storing information, the GDP system must also incorporate specific knowledge about the world in order to effectively manage the information. One aspect of this knowledge is the set of semantic constraints, which represent known properties of the physical world with which all data must be consistent. Examples of such constraints are:

- peaks are higher than the immediately surrounding terrain;
- lake boundaries are closed curves;
- rivers do not flow uphill; and
- geologic layers do not intersect one another.

Another aspect of world knowledge which should be incorporated in a GDP system is inference modelling, which describes how new information can be derived from existing information based on known properties of the physical world. Examples of such derivations include:

- the elevation between two known contour lines can be approximated via linear interpolation;
- the elevation of a point interior to a lake is the same as the elevation of the lake shore;
- if there are m missiles and s silos in area a, then the probability that a missile is in a given silo is m/s.

Such inference models add greatly to the power of the system both by providing a much larger (virtual) information set for the users and by allowing for automatic reinterpretation of information based either on new information or new models.

Given these concepts, the life cycle of information in the GDP system can be characterized as follows. First, information is acquired via the process described in the previous section and is added to the GDP system information store (subject to its meeting the established semantic constraints). Based on new information and on world models, the information may undergo several transformations and reinterpretations during the course of its existence. Information may also be superseded by new data which is deemed to be more complete or correct. Finally, information is deleted when it becomes irrelevant (i.e. it is no longer useful for any purpose and will not be accessed in the future).

2.1.5. Output Data Representation, Synth sis, and Transformation

The first step in the processing of output data from the GDP system is the synthesis of the necessary data from the GDP information store. The main object of the synthesis process is to obtain the data needed for the creation of a particular product or for the processing of input data. For the simpler applications the data may be extracted directly, but more complex applications may involve the generation of new information based on specialized models (oriented to a specific product or set of products). In addition, a general query capability is required for interactive systems. Typical criteria for information requests include position, shape or other topological attribute, various relationships between data, security classification, etc.

Once the output data has been synthesized, it is usually not in a form which is suitable for product generation. Hence, various transformation steps may be required before its suitability for product generation is achieved. Typical transformations include the extraction of specific information of interest, security screening, changes in coordinate systems to match the needs of the application, or other product-specific operations (such as viewpoint, scale, exaggeration, light source, and color scales as would be found in various cartographic products). Since different operations require different data formats to work efficiently, additional transformations are required in order to change data formats to meet the processing requirements.

2.1.6. Product Generation

Product generation is the process in which the GDP products are actually created. Some examples of products provided by GDP systems are:

- Two dimensional maps and charts.
- Three dimensional relief maps.
- Digital elevation matrices.
- Crisis support information.
- Weapons systems support information.
- Databases for other systems.
- Command, Control, and Communication system interfaces.
- Online query services.

Simulations are another class of products that make use of information from GDP systems. Within the framework presented above a simulation may be considered to be a sophisticated product generation process. Some examples of GDP related simulations are

- flood plain analysis,
- population evacuation,
- flight simulation,
- radar image simulation, and

- tactical simulations.

The final class of products generated by the GDP system are management reports. These reports deal with the status of the system itself, and aid in both current operation and planning. Two examples of status reports are coverage charts of geographic information and accuracy assessments of various information sets.

2.1.7. Non-Functional Constraints

In addition to the functional aspects of GDP systems presented above, there are a number of non-functional constraints common to most GDP systems which greatly impact their design and implementation. The major constraints which have been identified are described below; the implementation strategies necessary to design a system which meets these constraints will be discussed in other sections of this report.

One of the driving considerations in the design of a GDP system is the maximization of production throughput. From this standpoint, the GDP system is viewed as a tool for manufacturing various products. Since these products are the reason for the existence of the system, the volume of production and the efficiency with which it is performed are important measures for determining the appropriateness of a design. Hence, it is important to concentrate on both

(1) reducing the time required for both operating personnel and and processing equipment to perform their functions, and

(2) providing sufficient resources to handle the large volumes of information that are inherent in GDP applications.

A high level of robustness is another requirement for GDP systems. Hence, the system must have both the ability to accommodate failures of various system components and provide the capability to quickly isolate and fix problems within the system. This requirement is particularly important for those systems that are used for crisis support. A related requirement is that of survivability, which requires that a system be able to continue operations (or at least resume them quickly) in the face of disasters such as fire or sabotage.

Since GDP products are used in many operations which require a high level of accuracy, the quality of these products must be strictly controlled. Part of this can be accomplished by maintaining an assessment of the accuracy of each item of information maintained by the system. Since much of the information available is derived, however, this requires that models for accuracy assessment be included in the derivation process. Quality must then be maintained via careful structuring of the system to allow for checks and corrections at each step of the system processing.

The last major constraint we will deal with is security. Since much of the information in a GDP system is highly sensitive from a security standpoint (both for private corporations and government), protection of the information is required at all stages of acquisition, product generation, and processing in between. Although the security model required depends upon the organization for which the system is designed, a typical scheme would involve multiple levels of security with some sort of compartmentalization of information access. Some examples of the criteria by which access to information can be determined include:

- the person accessing the information;
- the function accessing the information;
- the location from which the information is accessed;
- the geographic area to which the information is related;
- the resolution or level of detail of the information.

The security attribute of each item of information must be either associated with the information directly or derivable from relationships with other information. Hence, the models for deriving new information from that already in existence must also generate the necessary security attributes (for example, feature information extracted from secure data may remain at the same level of security or greater, but statistical information based in part on secure data may be given a lower security rating).

2.2. RELEVANT TECHNOLOGY

2.2.1. Overview

The purpose of this section is to identify the areas of technology which are relevant to the design and implementation of a Geographic Data Processing (GDP) system. The first step in this identification is the establishment of the critical design issues and constraints based on the GDP requirements presented in the previous section. Then the areas of technology which address these critical design constraints and which have potential application in GDP systems are identified.

We consider GDP systems from the standpoint of their use as a production tool. Hence, the focus is on technology which will improve the overall productivity of the system with respect to:

- throughput in product development and generation;
- accuracy, completeness, and quality of products;
- responsiveness to changes in the nature and volume of production requirements;
- low maintenance and retooling costs;
- robustness and survivability; and
- security.

When possible, currently available technology which meets the GDP requirements is identified. Specific areas of research are identified, however, for those areas where current technology is either insufficient or continued technological development is necessary for the successful long-term operation of a GDP system.

The identification of the technology which is necessary for the implementation of a GDP system is not sufficient, as the complexity of GDP systems and the severe constraints which they must meet make the ad hoc application of available technology infeasible. A well organized and complete GDP design methodology is therefore a necessity for the successful development of a GDP system. Such a design methodology consists of a description of the steps which must be taken in order to design and develop a system, a definition of the formal specifications that are to be produced at various points in the design, and a description of the method and criteria for evaluating the design produced. The issues related to the development of a GDP design methodology will be treated later in this report.

The following presentation is divided into subsections based on the subdivisions of the GDP requirements model:

- data acquisition;
- input data representation, analysis, and transformation;
- management of factual data and world knowledge;
- output data representation, synthesis, and transformation;
- product generation.

In each of these subsections the design issues relevant to the requirements area are presented, and then the areas of technology which address these design issues are listed.

Figures are used to summarize the information.

2.2.2. Data Acquisition

See Figure 2.2-1.

Design Issues

(1) GDP systems require support for a large and constantly changing set of data sources and formats.

(2) The volume of data that is processed during acquisition is very large.

(3) The system must be capable of detecting and handling changing levels of accuracy, completeness, and quality of the input data.

Relevant Technology

(A) The use of virtual interfaces and devices is commonly used to allow a system to support a large and changing number of input formats and devices. . Examples of the use of virtual interfaces include the virtual graphics devices of packages such as the Graphical Kernel System (GKS), and the unified device and file I/O structures found in UNIX.

(B) Automatic digitization of source data allows for the rapid and efficient collection of large amounts of data.

(C) Inventory management systems can be used to keep track of the large quantities of data that must be processed as well as the quality of the data.

2.2.3. Input Data Representation, Analysis, and Transformation

See Figure 2.2-2.

Design Issues

(1) Varied data representations must be supported to meet the needs of the transformation algorithms that must be applied to the data.

(2) A large repertoire of transformations must be supported and developed based on the type of input source and the information which is to be extracted.

(3) Most input data sets are very large, making efficient and rapid processing of the data one of the most critical design constraints.

(4) Care must be taken to ensure that the input data are consistent with

	Productivity	Responsiveness to Change	Low Maintenance Cost	Varied Data Sources	Large Data Volume	Variable Data Quality
Virtual Interfaces		X	X	X		
Auto. Digitization	X				X	
Inventory Management					X	X

Figure 2.2-1. Data Acquisition

	Productivity	Product Quality	Responsiveness to Change	Low Maintenance Cost	Robustness	Security	Varied Representations	Varied Transformations	Large Data Volume	Data Consistency
Formal Data Models		X					X			X
Trans. Algorithms	X						X	X	X	X
Image Processing	X	X								
Feature Extraction	X	X								
Quality Estimation		X								X
Parallel Processing	X				X				X	
Fast Processors	X								X	
Mass Storage	X								X	
Flexible Structures			X	X						
Human Engineering	X	X								
Secure Comm.						X				

Figure 2.2-2. Input Data Representation, Analysis, and Transformation

information already existing in the GDP system.

Relevant Technology

(A) Formal data models based on the input source can be used to detect and correct source-related errors automatically. Examples of this are the models of remote sensor scan patterns that allow for the correction of distortions, and models of digitizing equipment which allow for the correction of distortions in the digitization process.

(B) Improvements and additions to the set of transformation algorithms will be required. Such algorithms include data error detection and correction, coordinate transformations, and the merging of new data with existing data.

(C) Image processing techniques are important in GDP systems, and their role will grow as new algorithms are developed and hardware capable of supporting the volume of data becomes available.

(D) Pattern recognition and feature extraction algorithms are an important means of gathering information from raw data and automatically determining such things as position and orientation of the data.

(E) Automatic quality assessment algorithms need to be developed.

(F) Parallel algorithms for performing the required computations and hardware capable of exploiting the concurrency of these algorithms need to be developed. The degree of parallelism available in current technologies is great, ranging from loosely coupled distributed processing systems (such as packet networks) to tightly coupled array processors or pipelined processors. The development of algorithms that can take advantage of the available parallelism to speed up the computations and improve system reliability are important for the future success of GDP systems.

(G) High speed processing technology will need to be exploited to handle the volume of computation required in GDP systems. Both exotic processor architectures (pipelines, processor arrays, special purpose VLSI circuits) and high speed electronics (ECL, GaAs, optical switches, superconducting switches) will be needed to meet the speed requirements.

(H) Mass storage devices that work at high speeds and handle large volumes of data will also be required. Some possible technologies include laser disks, magnetic bubble memories, and charge-coupled devices.

(I) Since GDP systems must be flexible enough to incorporate new technology and adapt to changing production needs, the system architecture must be designed to allow for change and evolution. Some techniques which enhance the flexibility of systems are: virtual system interfaces, which allow for reconfiguration of the external environment of the system without requiring massive alterations of hardware and software; open interconnection strategies which allow parallel systems to grow and shrink as needed to meet production requirements; and dynamic allocation of tasks to computing resources to allow for simple reconfiguration of the system computation structure on-the-fly.

(J) In order to maximize the productivity of the interactions of the GDP system and its users, well engineered human interfaces will be required. Technology which will be required to implement these interfaces includes conceptual modelling of user/system interactions, task analysis techniques, empirical evaluation criteria, information rendering and perception models, and specialized display hardware.

(K) Secure communications technology will be needed to protect GDP information in a distributed processing environment. Some example technologies are the access control mechanisms found in many operating systems (such as MULTICS, UNIX, or VAX/VMS), encryption algorithms to protect information in the event of unauthorized system access, and transmission line analysis techniques that monitor the physical integrity of the communications media.

2.2.4. Management of Factual Data and World Knowledge

See Figure 2.2-3.

Design Issues

(1) The system must be able to adapt to the changing relationships between information sources and the products generated from them.

(2) Massive volumes of data must be maintained by this function, as it manages all of the data that has been input to the system.

(3) Many different forms and types of data must be managed. Included among these are geometric types (point, line, surface, volume); line, surface, and volume variables; and both discrete and continuous data values.

(4) Temporal and probabilistic attributes for data must be maintained in order to model uncertainties and changes in information.

(5) The ability to manage redundant data and resolve the potential inconsistencies is required.

(6) The system must be capable of incorporating new world models and reinterpreting information based on these new models.

(7) The system must be capable of managing many different views and models of the data at the same time.

	Productivity	Product Quality	Responsivness to Change	Robustness	Security	Massive Data Volume	Source/Product Changes	Varied Data Types	Temporal, Probab. Info	Data Redundancy	World Models	Multiple Views of Data	Unanticipated Queries	Semantic Integrity
Secure Communications					X									
Artificial Intelligence	X	X	X								X	X		
Database Systems	X			X		X								
Database Models	X	X	X	X	X		X	X	X	X	X	X	X	X
Storage Heirarchy	X		X	X										
Mass Storage Devices	X					X								

Figure 2.2-3 Management of Factual Data and World Knowledge

(8) The system must be able to handle arbitrary queries which may not have been anticipated by the designers.

(9) The system must at all times maintain the semantic integrity of the information and models within it.

Relevant Technology

(A) As always, secure communications technology will be needed to protect any information which must be transported between nodes in a distributed database or between the GDP system and its points of access.

(B) Artificial intelligence capabilities must be incorporated into the system to support a number of database activities. The relevant areas include intelligent query processing, logical inference and deduction systems, and knowledge based (expert) systems.

(C) Sophisticated technologies will be required in order to implement the database system chosen. These technologies include distributed database architectures, database processors, and associative mass storage devices.

(D) Sophisticated database models will be needed in order to handle both the raw information and the world models via which new information is derived. Previous studies indicate that the standard models (such as the relational, hierarchical, and network models) are not likely to be useful for managing a GDP database. Database models which are more likely to be useful within the GDP environment include the information systems models (such as the Abstract Database System), and logic-based systems.

(E) A hierarchical mass storage organization such as that found in the MULTICS operating system will be needed in order to manage the large quantity of data.

(F) High speed, high volume mass storage devices will be essential in order to achieve the response times needed.

2.2.5. Output Data Representation, Synthesis, and Transformation

See Figure 2.2-4.

Design Issues

(1) Multiple data representations and formats must be supported to allow for a diverse set of products and data transformations.

(2) As with the input processing, the GDP system must be capable of managing a high volume of data in order to meet the needs of the product generation.

	Productivity	Product Quality	Responsiveness to Change	Low Maintenance Cost	Robustness	Security	Varied Data Formats	One-of-a-kind Products	Changing Requirements	Large Data Volume
High speed computers	X									X
Mass Storage	X									X
Distributed Proc.	X		X		X					X
Secure Commun.						X				
Software Dev. Env.			X	X			X	X	X	
Query Languages			X					X	X	
Prob. Oriented Lang.			X				X	X	X	
Human Factors	X	X	X	X						

Figure 2.2-4 Output Data Representation, Synthesis, and Transformation

(3) The system must be flexible enough to adapt to the rapid changes in the information which will be required by the product generation process.

(4) The system must also provide the ability to develop one-of-a-kind products rapidly, as needed for crisis support or other special applications.

Relevant Technology

(A) High speed processors will be necessary.

(B) High speed, high volume mass storage devices will also be required.

(C) Distributed processing systems technology will be needed to support the volume of data and allow for fault-tolerant designs.

(D) Secure communications technology will be needed to protect any data that must be transmitted over publicly accessible communications media (phone lines or other transmission lines which pass through non-secure areas, microwave links, or satellite links).

(E) A software development environment must be provided for the system which will allow for the rapid development of new synthesis or transformation procedures to the system, as well as aid in the maintenance of existing procedures. Examples of the technology required include the Ada programming support environment (APSE), the Modern Programming Environment (MPE) at DMA, and the Total System Design (TSD) Methodology.

(F) Various query languages will need to be developed based on the particular application and means of access to the system.

(G) In order to facilitate the development of specific classes of synthesis and transformation procedures, a number of problem oriented languages may need to be developed for the GDP system.

(H) Well engineered human interfaces must be provided to maximize the efficiency of the man-machine interactions and thus improve the overall productivity of the synthesis and transformation processes.

2.2.6. Product Generation

See Figure 2.2-5.

Design Issues

(1) The system must be capable of supporting a large and constantly changing set of products.

(2) The system must be able to adapt to changes in the technology used to create products.

	Productivity	Product Quality	Responsiveness to Change	Low Maintenance Cost	High Volume of Data	Large Set of Products	Changing Production Technology	Growth of Soft Products	Unanticipated Questions	Robustness	Security
Auto. Generation	X	X				X	X				
Virtual Interfaces			X	X		X	X				
Human Factors	X	X							X		
Smart Workstations	X	X				X		X	X		
Prod/Inv Management	X	X	X		X	X	X			X	X

Figure 2.2-5 Product Generation

(3) The system must be flexible enough to manage both the generation of current "hard" products such as maps, and a growing set of "soft" products such as simulations and databases for other systems.

(4) Support for the rapid development of one-of-a-kind products and unanticipated queries to the system must be provided, particularly for C^3I systems applications.

(5) As in all other aspects of the GDP system, the large volume of data that must be handled is a principal design concern.

Relevant Technology

(A) Automatic product generation and evaluation technology will be needed to support changes in the production requirements. Examples of such technology are the problem oriented languages and applications generation packages.

(B) Virtual system interfaces will be required to support changes in the nature and configuration of the devices used to generate products.

(C) Human factors engineering will have to be applied to the interactive product generation interfaces in order to maximize the overall productivity of the generation process.

(D) Intelligent workstations which can control the individual product generation processes will be needed to achieve the throughput requirements and enhance the robustness of the system.

(E) Production control and inventory management systems will be needed to handle the logistics of operating an efficient production facility.

2.3. PRESSING R&D NEEDS IN THE GDP AREA

2.3.1. Overview

The areas of technology which are relevant to GDP systems were identified in the previous section. Due to the extreme requirements placed on GDP systems (system size, volume of data, production throughput, security, etc.), current technology is unable to meet all of the needs of such a system. Research is therefore required in several technological areas in order to make feasible the development and operation of GDP systems that will meet anticipated production needs. Among the various areas of technology reviewed in Section 2.2, those that have a high immediate payoff or which require immediate attention for research and development are identified here. Research areas pertaining to technology required for GDP systems themselves are presented in Section 2.3.2, and technology needed in order to carry out the design of GDP systems is presented in Section 2.3.3.

2.3.2. GDP System Technology

Image Processing

Images constitute a major input medium for GDP systems. Diverse devices such as stereo photographic systems and satellite sensors produce image data that must be processed in the GDP system. In order to prepare these images for analysis and improve their quality, image processing techniques will be required. Typical operations performed currently in image processing include correction of distortions from the imaging device; normalization of the image to a standard scale, orientation, and resolution; and the enhancement of various features of interest so that later problems of image analysis are simplified.

Work has already been done in a number of areas. Some of the simpler image transformations include the rectification and enhancement of grey scale or color according to some model of the input sensor. More complex image transformations include scaling and orientation of images to conform with some standard. Image filtering is a technique which is used to prepare images for later analysis—filters can be used to enhance features such as edges in order to simplify region identification and texture analysis. Pattern matching is also applied to image analysis, and is used in applications such as the analysis of stereo photograph pairs.

Although much existing work is directly applicable to solving GDP image processing problems, additional research is needed. In particular, the large volume of images that require processing necessitate the development of parallel algorithms and specialized hardware to support these operations. Additional research into sensor models to be used for image rectification will also be necessary as new image producing devices are added to the system.

Image Analysis

As noted above, a large quantity of GDP input data is in the form of images such as photographs, maps, and remote sensor data. Under current technology, a great deal of manual effort is needed to perform such tasks as the identification of the position and orientation of the image, the identification and classification of features within the image, and the tracing of contours and profiles. In order to meet the high throughput and quality requirements of GDP systems, automatic image analysis techniques must be researched and adapted to GDP problems.

Existing work has dealt with a broad spectrum of problems ranging from the development of basic geometric analysis techniques to the use of knowledge about the world to guide analysis of images. Typical geometric analysis techniques include edge/boundary detection, segmentation of an image into regions, and 2-D pattern matching and pattern recognition. In addition to the identification of geometric structures in a image, techniques for representing these structures in a manner best suited for analysis have been developed. Since various features of interest in complex environments are not easily identifiable on the basis of their geometry alone, the use of knowledge based systems to guide the analysis of images has also been explored. These systems (some based on logic models, some on semantic nets) work to identify the content of an image based on models of properties that objects in the image domain should possess and the relationships of objects to each other.

The following research areas need to be pursued in relation to image processing in order to make its use practical in GDP systems:

(1) Current work in image processing must be adapted to deal with GDP-type problems.

(2) Specialized high-speed hardware and parallel algorithms for processing images must be developed in order to support the analysis of images on a production basis.

(3) Systems and models for representing knowledge within a GDP environment must be developed.

(4) Techniques for using knowledge from the GDP database to aid in the image analysis process must be researched. Uses of this include the placement of a new image in an existing context within the GDP database, the use of existing redundant data to aid in the analysis of images (such as through the identification of known landmarks), and the determination of various temporal properties of objects identified in the images.

Specialized High-Speed Processing

The high volume of information which must be processed in a GDP system places demands on the system which cannot be met by current technology. One critical approach to solving this problem is the development of algorithms and specialized hardware that are capable of performing their functions at very high speed. For many problems the use of high speed processors and I/O equipment may be sufficient (and perhaps available under commercial technology). Other problems such as image analysis will require the development of massively parallel algorithms and parallel hardware in able to perform these tasks within a production environment.

A great deal of work has already been done in support of applications such as weather forecasting, wind tunnel simulations, geologic data processing, and very large database systems. In the area of parallel computer architectures, a broad spectrum of architectures have been postulated (although few have been built). Typical structures include processor arrays (designed for executing tightly coupled parallel algorithms), pipeline architectures (for rapid serial vector processing, as in the CRAY computers), clustered machines (designed for loosely coupled concurrent processing organized around a shared file system or database), and computer networks (supporting distributed processing requiring minimal process interaction). Another area of current technology which may be important in GDP systems is the capability of developing custom VLSI devices for performing specialized tasks at very high speed. Relatively simple designs can be done using gate array technology, which allows for logic designs to be implemented directly. More intricate designs can be made using mask definition languages and associated circuit layout design tools in order to specify a custom chip design. Large, complex circuits may be designed with the aid of automatic chip layout and simulation systems (as was done with the Bell Labs echo suppressing chip). New device technologies which may provide the processing speeds necessary for GDP applications are currently being researched. These include gallium-arsenide field-effect transistor (FET) devices (currently used primarily in microwave systems, although some digital applications are emerging), superconducting switches, and optical switching and communication systems.

Research in this area that must be done in support is GDP systems includes:

(1) The development of parallel architectures that are oriented toward the efficient processing of various GDP functions (such as image processing and analysis, database processing, and product generation).

(2) The identification of those functions and algorithms in GDP systems that are amenable to implementation as special-purpose VLSI hardware.

(3) The incorporation of the new high-speed device technologies in special applications.

Database Modelling

A unified model of a GDP information structure serves as a conceptual framework around which the entire GDP system can be organized and managed. This model will determine the nature of the data maintained by the GDP system, and thus will have an impact both on the ease of use of the system and the efficiency of data manipulation within the system. Recent work has indicated that the "standard" data models (network, hierarchical, and relational) are not well suited for most GDP applications.

Hence, research into the types of data models that will be most effective for GDP applications is necessary.

Most recent work in the data modelling area has been directed toward the relational data model. Specific areas of focus include the development of a formal mathematics of data models, data model design and analysis, methods for efficient implementation of database functions, the incorporation of artificial intelligence techniques (both for efficiency of implementation and efficiency of the user interface), and the engineering of human interfaces (particularly query languages). Additional work has been done in using systems of logic as a basis for data modelling. These infological models appear to be very promising as a means of building flexible, powerful, and efficient database systems.

Critical areas of research in the data modelling area are as follows:

(1) The development of data modelling techniques needed for supporting image processing applications.

(2) The incorporation of temporal and probabilistic information within the GDP data model.

(3) The incorporation of logical inference capabilities into the model.

(4) The coordination of the data models with the information models used in the GDP requirements definition.

(5) The compatibility of the data model with the models used in other artificial intelligence systems within the GDP system.

Database Implementation

The large volume of data and the high throughput requirements in GDP applications greatly exceed the capabilities of conventional database implementation schemes. Most current work on very large database systems, based mainly on relational database models, is not likely to be directly applicable to the GDP application area. Research will therefore be needed on new database implementation technologies oriented toward the demanding requirements of GDP systems.

Existing work which is not model-specific deals mainly with improving the speed and efficiency of low level data access and manipulation problems. One approach has been the use of "backend" processors which are separate from the user applications processor and which are designed to carry out low level database operations efficiently. Another approach to speeding up search times within a database is the implementation of associative mass storage devices via disk caches and processor per track devices. Extremely large quantities of data have been handled via hierarchical storage management systems, in which data is migrated between various levels of storage from main memory (fast access, but a scarce resource) to archival storage such as tapes (plentiful, but with very long access times). These systems attempt to move data between levels in such a way as to minimize the average data access time. A last approach is to use planning systems to organize data accesses in order to minimize the amount of searching needed to satisfy information requests.

The areas of research which need to be pursued in order to make the implementation of a GDP database feasible are as follows:

(1) Determination of the implementation technologies that are best suited to the data model developed for the GDP system.

(2) Techniques for managing hierarchical storage systems to minimize the average data retrieval time. A particular area of interest is the development of criteria for moving data from one storage medium to another.

(3) The possibilities for using new mass storage technologies (such as optical disks) for either direct access or archival storage within the storage hierarchy.

(4) Analysis of the tradeoff between integration of the entire GDP database system and degradation of performance due to the massive volume of data.

Security Models

Due to the sensitive nature of the information contained in GDP systems, provision must be made in the design of the system to protect that information from unauthorized access and modification. In order to ensure that all of the possible means of accessing protected information are eliminated, a thorough approach to security based on a unified security model must be taken. The incorporation of a security model into the GDP system will provide the means by which the system security can be organized and controlled.

Due to the pervasive need for security both in the government and in private industry, a great deal of work has been done in this area. Operating system security has been a major focus, concentrating on the protection of system resources from use or modification by unauthorized users. In particular, the concepts of security attributes for objects within the system and security capabilities for users have come from this area. Other work has dealt with protecting information which must be transmitted within a public environment (such as in the ARPANET). A third area which will be of importance to GDP systems deals with security issues in statistical database query systems. This work deals with preventing the discovery of protected information in the statistical database via a set of otherwise valid and seemingly innocuous queries. Lastly, some work has been done on developing formal models of multi-level, compartmentalized security schemes such as those found within the DoD.

Research needs in the security area are as follows:

(1) Development of a formal model for GDP security, based on the concepts of levels and compartmentalization.

(2) Development of methods for determining the security attributes which are to be assigned to system users (such as what constitutes a need to know).

(3) Development of methods for automatically assessing the security attributes to be attached to data on input.

(4) Research into models for determining the security attributes to be assigned to information which has been inferred from other data in the system (such as in intelligent query systems) or which represents some summary of secure data (e.g. mean population density may be less sensitive than raw census figures).

Human Factors

Although many of the functions within a GDP system will be automated, some of the activities in the system must involve the participation or supervision of system users. Since these activities may be critical to the overall function of the system, the human interfaces must be as efficient as possible in order to maximize the productivity of the system. Various aspects of the interface which must be taken into account include the time for the user to carry out his tasks, the stress that the interface places on the user over a period of time, the amount of training time necessary before the user can perform his task efficiently, and the quantity of system resources necessary to support the user interaction.

Current work in human factors has tended to focus more on the evaluation of existing systems rather than on establishing human factors oriented design principles. Evaluation techniques are based on concepts such as user satisfaction and user performance (e.g. time to learn the system in question or time to perform a set of tasks using the system). Attempts have been made to determine the most effective way to render information to enhance user understanding, borrowing many ideas and concepts from the field of perceptual psychology. Some techniques more suited for use in the design process deal with the modelling of the users' interactions with the system (usually based on some sort of finite state machine or petri net model), and the modelling of the users' concept of the system's functionality.

The research needs in the area of human factors are as follows:

(1) Development of production time-oriented models and measures for user/system interactions.

(2) Analysis of the tasks to be performed by users in order to locate bottlenecks and points of confusion.

(3) Development of conceptual models of human/system interactions. These models should be consistent with the models developed for the artificial intelligence and database systems.

(4) Development of information rendering and perception models.

(5) Engineering of the interaction media (displays and input devices) to facilitate user/system interaction.

Product Specification Techniques and Standards

As in current CAD/CAM systems, the use of product specifications to guide the product generation process will make a GDP system more flexible (changes in production can be implemented by simply changing the specifications) and decrease the time necessary to develop a new product (which is critical for unanticipated crisis support situations). Although the need for this is apparent with the current set of "hard" products (such as maps) that predominate in GDP systems, the increasing demand for "soft" products (such as simulations and special-purpose databases) will demand a flexible means for specifying how these are to be generated from the GDP system.

Current work in computer aided design and manufacturing systems is largely applicable to GDP systems. Current research into techniques for developing and implementing problem oriented languages (POLs) will also be useful in the development of GDP-oriented POLs. Lastly, expert query systems may play a major role in helping the user specify the products that he wants.

The research needs in the area of product specifications are as follows:

(1) Design of langauges which are suitable for the specification of a wide variety of GDP products.

(2) Development of query systems for extracting information directly from the GDP databases. These query systems may involve a wide range of human interfaces such as voice input and graphics displays.

(3) Development of expert systems oriented towards specific applications areas (such as geology, demographics, troop movement and logistics, navigation, etc.).

2.3.3. GDP Design Technology

In order to exploit the available technology effectively in the implementation of a GDP system, methodologies that promote and exercise control over technology usage are needed. Methodologies for the development of GDP systems must be specially tailored based on the nature of the GDP problem area, the structure of the organization which will be using the GDP system, and the areas of technololgy which will be incorporated in the system.

A complete system development methodology includes methodologies for problem definition, system design, and system implementation. The research areas relating to the definition of the complete requirements for a GDP system are presented in Section 2.3.3.1. Those pertaining to the definition of a GDP design methodology are presented in Section 2.3.3.2. Software design and development issues (in particular the impact of

Ada in these areas) are presented in the Section 2.4.

2.3.3.1. Problem definition

The overall purpose of the problem definition process is to develop a formal definition of the GDP system requirements which will be used as the basis for the GDP system design and provide the means for validating that design. In the development of this definition, the following goals are considered to be important for GDP systems:

- Emphasis of facts rather than raw images;
- Separation of the concept of information contained in data from the format of the data itself;
- Support for human factors evaluation techniques such as conceptual modelling of the user view of the system, task analysis, and rapid prototype generation;
- Support for the concept of expert systems;
- Integration of the data models used in requirements definition and in database definition.

Initial research has indicated a number of properties which the GDP requirements definition should possess. With respect to data definition, the most fundamental characteristic is the reduction of all information to a set of logical facts. The introduction of a set of semantic domains including time, space, security, and probabilistic qualifiers to facts is also important. The model must provide for the definition of semantic constraints over data, as these constraints will aid greatly in the preservation of data consistency. Lastly, the model should acknowledge the biases in the different views of data held by various applications, and allow these applications to maintain their own views and models of the information.

The data extraction concepts contained in the definition should also possess a number of characteristics. First, support must be provided for an extensive set of extraction criteria (including geographic, topologic, geometric, temporal, and via inference). Support must also be provided for human engineering considerations, particularly with respect to the flexibility of data views and the types of query languages that can be supported. Lastly, the definition should be suitable as a basis for the development of expert systems which will use the knowledge contained in the system data.

The last set of model characteristics identified deals with the operations that are defined on the data. The model must provide for an arbitrary set of transformations to be made on the data, which are viewed as merely changing the format of the data but not necessarily the information content. The model must also support inference capabilities, allowing for new information to be derived from existing information based on models of the world in which the information exists.

In order to carry out the full definition of the GDP requirements, a requirements definition and verification methodology must be developed. This methodology must deal with four areas: system data modelling and evaluation, definition of system functions, human factors evaluation, and performance constraints clarification. The definition of the methodology will be based on meeting the goals and characteristics for the requirements model that were presented above.

2.3.3.2. System design

In order to carry out the system design in an effective manner, a formal methodology must be defined and developed. The methodology developed should meet the following goals:

- The design should be placed in a total systems perspective, in which hardware and software are treated on an equal footing.

- A systematic approach to the design should be emphasized. This is particularly important for GDP systems, as their cost and importance make an ad hoc approach to design unacceptably risky.

- The methodology must give the designer freedom to employ new technology but also maintain control over the application of this technology.

- Due to the great expense of redesign after the fact, the methodology must ensure that system meets the specified requirements at each step in the design process.

- The methodology should help identify the system development tools that are needed to decrease development costs, increase productivity, and improve the quality of the system produced.

The Total System Design (TSD) methodology is one methodology which is designed to meet the goals described above. This methodology incorporates a number of concepts that are extremely important in dealing with systems of the size and complexity of GDP systems. In the TSD methodology, system designs are structured as a hierarchy of virtual machines, a technique which has proved useful in dealing with large, complex systems. System design proceeds from the top down, with a dual stepwise refinement of the hardware and software specifications for the system (encouraging the exploration of hardware/software tradeoffs). The methodology also requires integration of the functional and performance requirements for the system. These requirements are used both to drive the design of the system and as a means of verifying the correctness of the system design at each step in the development process.

In order to be used effectively for GDP systems, the TSD methodology must be tailored to the specific problems and areas of technology that will be encountered within a GDP environment. This tailoring involves the selection of appropriate GDP oriented specification techniques, evaluation strategies, and methods for verifying the design against the GDP requirements. Particular areas of focus in the development of this GDP/TSD methodology include:

- the specification and analysis of locally distributed systems;
- the use of high speed special purpose processors;
- the use of graphics as an input/output medium;
- the organization of databases in the face of massive quantities of data;

- the tradeoff of system integration versus overall performance;
- the development of system structures that are appropriate for the effective application of new technology.

(Note: Although GDP applications will likely require high speed special purpose hardware, we do not envision that DMA will engage in the development of its own hardware systems. Instead, we believe that DMA will continue to have this development performed by outside contractors.)

2.4. NEAR-TERM ADA IMPACT

The objective of this section is to outline a near-term strategy for the assimilation of the Ada language in the DMA production environment. The opportunities and the challenges offered by Ada are reviewed first. This leads to a discussion of the Ada training needs at DMA. A sample training program is illustrated in the section on the development of an Ada culture within the organization. Finally, we put forth a proposal for a low risk strategy leading to the integration of Ada into the DMA production environment.

2.4.1. Ada—Opportunities and Challenges

In the context of the generally acknowledged software crisis, Ada attempts to provide some of the answers. Without being comprehensive in scope, this is the very first language whose motivation rests with methodological principles that emerged form the failures and successes of programming in the large during the past decade.

All key technical features characterizing Ada provide immediate support for some recognized software engineering need:

- task—real time programming, concurrency, parallelism
- package—maintainability, flexibility, etc.
- separate compilation—modular development
- data abstraction—clarity, maintainability
- strong typing—reliability
- generics and task types—component reusability
- programmer defined exceptions—reliability
- representation specification—real time programming

Without being exhaustive, this list clearly illustrates the potential significance of the use of Ada at DMA.

The challenges are basicly two: personnel conversion and production conversion. The former is made difficult by the general lack of formal computer science training of many computer professionals to date and by the complexity of the Ada language. Although Ada designers deliberatly limited the language features to those that captured proven concepts, these concepts are novel for the average practitioner. The latter is complicated by the large investments in earlier outdated languages and software technologies. It is our intent to demonstrate in the next three subsections that DMA is in a position to meet this challenges without major disruptions in the production process.

2.4.2. Training Requirements

Making effective use of Ada requires a highly intensive training program that must cover, at a minimum, three areas of technical expertise:

- Ada language features and their use.
  Since most often it is not feasible to precede the Ada language training by traditional computer science courses, it is important to cover not only the language features but also the principles on which they are based. Control structures, data structures, storage management, scope rules and concurrency are some of the topics that should be included.

- Ada related software engineering methodologies.
  Good language skills are not sufficient to exploit the opportunities created by Ada. Moreover, using Ada to code FORTRAN programs will be of no service to the organization. The language alone does not eliminate the complexities of software design. (One may be able to read circuit diagrams without knowing how to design circuits.) The introduction of Ada related software engineering methodologies, tuned to the needs of the organization and its application area, must take place before the first project fails and renders the language as the scapegoat.

- Ada programming support environment.
  While the methodology contributes to software quality, maintainability, and managibility, the use of tools is expected to result in significant productivity increases. To achieve this, an understanding of how to use the tools effectively in the context of the methodology is required.

  Another important technical area is that of requirements specification. Since it is not necessarily Ada-specific, it will be ignored it in this discussion.

2.4.3. Developing an Ada Culture

Effective use of Ada demands the establishment of a new cultural milieu within the organization, a novel approach to software design and maintenance. Here are some steps that may be taken in pursuit of this objective:

- ADA APPRECIATION COURSES.
  Short seminars provide a vehicle for introducing the audience to the Ada language (motivation, concepts and features) and its impact on software engineering practices. A seminar series covering the key features of the Ada language is valuable for an audience of programmers as a starting point in the study of Ada. This series is most effective when combined with a programming workshop.

- ADA-BASED DESIGN LANGUAGE AND METHODOLOGY.
  This option is intended for organizations that want to develop an Ada culture through the use of Ada related software engineering technology on projects where the implementation language continues to be some other

existing programming language. This may be accomplished by formulating the methodology around a design language that makes use of a subset of Ada compatible with current implementation languages used by the organization. ADL, developed by G.-C. Roman and McDonnell Douglas, is one example of such a design language—it assumes FORTRAN and assembly language to be the target implementation languages and real time embedded systems to be the application area. The training must include the programming techniques required to implement, in a cost effective manner, the selected Ada features. In this manner, modern design/programming technology and Ada compatible documentation may be adopted without disrupting the organization as an interim preparatory step to introducing the Ada language.

- COMPREHENSIVE ADA LANGUAGE AND SOFTWARE ENGINEERING TRAINING.
  The objective is to provide an understanding of the fundamental and unique features of the Ada programming language and of the software engineering techniques required to use effectively Ada in the development of software systems.

Considering the last option in more detail, we provide below a listing of the subject matter that ought to be covered by a comprehensive training program.

LANGUAGE ISSUES.

Programming Language Fundamentals
Program structure, lexical structure, data structure, control structure, sample programs.

Data Types
Motivation; scalar types, constraints and subtypes, derived types, array types.

Package Cnncept
Motivation; abstract data types; private types and limited private types; complex data structures: record and access types.

Ada and Programming in the Large
Visibility rules; generic packages; separate compilation; exception handling.

Tasking
Motivation; task types; task execution; rendezvous; selective waits; timed entry calls and accepts; interrupts; exceptions.

METHODOLOGY ISSUES.

Software Engineering Fundamentals
Specification, design and implementation; abstraction and levels of abstraction; procedural and non-procedural specifications; modelling.

Structured Programming

Procedure/function specification and body—requirements versus design; requirements specification: input/output assertions, tables, etc.; design techniques: stepwise refinement, recursion, fault tolerance, etc.; design specification: Ada-based pseudocode.

Package Specification and Design

The economics of the package concept: information hiding, maintainability, expandability, flexibility, interchangeability of parts, etc.; package specification: abstract information structures, axiomatic versus operational specifications, engineering and human factors; package design: data structures design and algorithmic efficiency.

Object-oriented Design

Comparative study of program organizations and design methods; modularity, virtual machine hierarchies, simple object oriented structure, vertically-structured object hierarchies, resource hierarchies; top-down design, bottom-up design, critical path design, data structure design, structured design, design evaluation methods; implementation/testing strategies.

Real-time System Design

Concurrency and tasks; task scheduling; communication: message-based, shared-data via shared packages; anomalies (deadlock, live-lock, unfair scheduling, etc.) and concurrency coordination; factors influencing the *selection of a community of tasks:* modularity, timing and space requirements; studing the system's behavior and performance characteristics by modelling the software architecture using Ada; a sample object-oriented design methodology for real-time systems.

2.4.4. Integrating Ada into the DMA Production Environment

DMA is currently in the process of introducing a Modern Programming Environment (MPE). The key to successful introduction of Ada is to use the MPE as a stepping stone by following the strategy below:

- use the MPE to introduce the organization to the widespread use of tools and standardized methodologies;

- require MPE compatibility with some Ada Programming Support Environment (APSE);

- exercise care in the development of the component library postulated by current MPE design proposal;

- identify which aspects of the MPE (tools, human factors, etc.) have the greatest impact on productivity and software quality;

- acquire APSE compatible with the MPE;

- enhance MPE approaches with Ada concepts and design strategies like the ones outlined in the previous subsection;

- introduce limited Ada usage;

- using the MPE library, build packages by employing the following two approaches

  - identify general application concepts that emerged from the building and use of the MPE library and develop Ada packages that would make their standard definitions and related operations available for future application programs

  - identify highly used and costly to recode FORTRAN programs and develop standard Ada package specifications of these programs while keeping the package body in the original language (This is made possible by the use of the pragma INTERFACE.)

- enhance APSE to include successful features of the MPE;

- introduce large scale use of Ada.

2.5. SELECTED BIBLIOGRAPHY

[BALL82] Ballad, Dana H., and Brown, Christopher M., *Computer Vision,* Prentice-Hall, 1982.

[CARN71] Carnap, Rudolf, and Jeffrey, Richard C., editors, *Studies in Inductive Logic and Probability,* University of California Press, 1971.

[CAST81] Casto, T. L., and Bell, D. M., *Universal Rectifier System Study* , Vol. 2-3, Harris Corporation, 1981.

[CHAN81a]
Chang, N. S., and Fu, K. S., "Picture Query Languages for Pictorial Data-Base Systems", *Computer,* IEEE, November, 1981.

[CHAN81b]
Chang, Shi-Kuo, and Kunii, Tosiyasu L., "Pictorial Data-Base Systems", *Computer,* IEEE, November, 1981.

[CHI82] Chi, Chao S., "Advances in Computer Mass Storage Technology", *Computer,* IEEE, May, 1982.

[CHOC81] Chock, Margaret, Cardenas, Alfonso F., and Klinger, Allen, "Manipulating Data Structures in Pictorial Information Systems", *Computer,* IEEE, November, 1982.

[CLAI82] Claire, Robert W., and Guptill, Stephen C., "Spatial Operators for Selected Data Structures", *Proceedings of Auto-Carto 5,* American Society of Photogrammetry, 1982.

[CLIF83] Clifford, James, and Warren, David S., "Formal Semantics for Time in Data Bases", *Transactions on Data Base Systems,* ACM, June, 1983.

[DENN79] Denning, Dorothy E., and Denning, Peter J., "Data Security", *Computing Surveys,* ACM, September, 1979.

[FAIR82] Fairborn, Douglas G., "VLSI: A New Frontier for Systems Design", *Computer,* IEEE, January, 1982.

[HAYN82] Haynes, Leonard S., Lau, Richard L., Siewiorek, Daniel P., and Mizell, David W., "A Survey of Highly Parallel Computing", *Computer,* IEEE, January, 1982.

[HIRS82] Hirsch, Stephen A., and Glick, Barry J., "Design Issues for an Intelligent Names Processing System", *Proceedings of Auto-Carto 5,* American Society of Photogrammetry, 1982.

[HOLM82] Holmes, Garry L., "Computer Assisted Chart Symbolization at the Defense Mapping Agency Aerospace Center", *Proceedings of Auto-Carto 5,* American Society of Photogrammetry, 1982.

[HONA82] Honablew, Joseph A., Schlueter, Joseph J., Noma, Authur A., "Software for Three Dimensional Topographic Scenes", *Proceedings of Auto-Carto 5,* American Society of Photogrammetry, 1982.

[HWAN83] Hwang, Kai, and Fu, King-sun, "Integrated Computer Architectures for Image Processing and Database Management", *Computer,* IEEE, January, 1983.

[KIMU82] Kimura, Takayuki D., Gillett, Will D., and Cox Jr., Jerome R., "Abstract Database System (ADS): A Data Model Based on Abstraction of Symbols", Technical Report, Computer Science Department, Washington University, St. Louis, July 1982.

[LAND81] Landwehr, Carl E., "Formal Models for Computer Security", *Computing Surveys,* ACM, September, 1981.

[LEMP79] Lempel, Abraham, "Cryptology in Transition", *Computing Surveys,* ACM, December, 1979.

[MAGU82] Maguire, Martin, "An Evaluation of Published Recommendations on the Design of Man-Computer Dialogues", *International Journal of Man-Machine Studies,* Academic Press, 1982.

[MCLE80a]McLeod, Dennis, and Smith, John Miles, "Abstraction in Data Bases", *Proceedings of the Workshop on Data Abstraction, Databases, and Conceptual Modelling,* ACM, June, 1980.

[MCLE80b]McLeod, Dennis, "On Conceptual Data Base Modelling", *Proceedings of the Workshop on Data Abstraction, Databases, and Conceptual Modelling,* ACM, June, 1980.

[MINK78] Minker, Jack, "An Experimental Relational Data Base System Based on Logic", *Logic and Data Bases* (Edited by Herve Gallaire and Jack Minker), Plenum Press, 1978.

[MIRK82] Mirkay, Francis M., "Defense Mapping Agency Large Scale Data Base Integration", *Proceedings of Auto-Carto 5,* American Society of Photogrammetry, 1982.

[NAGY79] Nagy, George, and Wagle, Sharad, "Geographic Data Processing", *Computing Surveys,* ACM, June, 1979.

[PRES81] Preston, Jr., Kendall, "Cellular Logic Computers for Pattern Recognition", *Computer,* IEEE, January, 1983.

[REIS81] Reisner, Phyllis, "Human Factors Studies of Database Query Languages: A Survey and Assessment", *Computing Surveys,* ACM, March, 1981.

[RESC71] Rescher, Nicholas, and Urquhart, Alasdair, *Temporal Logic,* Springer-Verlag, 1971.

[ROSE83] Rosenfeld, Azriei, "Parallel Image Processing Using Cellular Arrays", *Computer,* IEEE, January, 1983.

[RUE82] Rue, Darryl L., "Computer-Assisted Feature Analysis for Digital Landmass System (DLMS) Production at DMA", *Proceedings of Auto-Carto 5,* American Society of Photogrammetry, 1982.

[SOWA82] Sowa, John F., "A Conceptual Schema for Knowledge-Based Systems", *Proceedings of the Workshop on Data Abstraction, Databases, and Conceptual Modelling,* ACM, June, 1980.

[STAN79] Stankovic, John A., and Van Dam, Andries, "Research Directions in (Cooperative) Distributed Processing", *Research Directions in Software Technology* (Edited by Peter Wegner), MIT Press, 1979.

[TSIC82] Tsichritzis, Dionysios C., and Lochovsky, Frederick H., *Data Models,* Prentice-Hall, 1982.

[ZOBR81] Zobrist, Albert L., and Nagy, George, "Pictorial Information Processing of Landsat Data for Geographic Analysis", *Computer,* IEEE, November, 1981.

3. SPECIFICATION AND VALIDATION OF GDP REQUIREMENTS

3.1. INTRODUCTION

We see Geographic Data Processing (GDP) systems as the key vehicle for automating significant parts of the DMA production process and thus reducing the high level of human labor currently required (particularly with regard to photo interpretation). Due to the extreme requirements placed on GDP systems needed by DMA (size, data volume, production throughput, etc.) and due to limitations in the current state of the art, these systems are expensive and difficult to design.

> THE GOAL OF THIS SECTION IS TO PROPOSE A REQUIREMENTS SPECIFICATION AND VALIDATION METHODOLOGY. THIS METHODOLOGY COULD REDUCE SIGNIFICANTLY THE RISK ASSOCIATED WITH BUILDING GDP SYSTEMS FOR DMA THROUGH THE DEVELOPMENT OF INEXPENSIVE RAPID PROTOTYPES. THESE PROTOTYPES CAN BE SUBJECTED TO TECHNICAL AND HUMAN FACTORS EVALUATIONS INVOLVING BOTH TECHNICAL EXPERTS AND PRODUCTION PERSONNEL.

Rooted in the notion of rapid prototyping, our strategy is to build simulations for the functionality of some proposed system, or part thereof, and to record intermediary results of the simulation for the purpose of an interactive real time play-back of the simulation script under user control. Built-in adjustable delays reproduce the performance characteristics of the technical solution under consideration. In this way both the functional adequacy and the potential production impact may be clearly identified prior to committing to the actual purchase or development of the system. The evaluators may be either technical experts or actual production personnel and the simulations may be used both as a means for requirements validation and for technical evaluation of proposed designs as well as an avenue for building and evaluating training materials for new production systems.

The remainder of Section 3 reports on the progress we have made toward the development of the GDP requirements specification and validation methodology. The presentation is divided into five parts:

Section 3.2 reviews the current state of the art in the requirements engineering field. The content of a requirements specification is presented in light of the consensus reached by both theoreticians and practitioners. The desirable properties of a requirements specification are justified from a functionalist viewpoint and it is suggested that changes in the way one uses the requirements may alter the relative significance of different properties. Finally, it is shown that, despite significant growth, the requirements area still faces a number of important unresolved issues including the need for a broader formal foundation for both functional and non-functional requirements, a greater degree of formality and automation, new requirements development methods and a higher level of integration in the overall design process.

Section 3.3 introduces the requirements specification and validation methodology (RSVM). It defines a systematic approach for evaluating proposed systems with respect to functional adequacy, technology utilization, performance characteristics, and human factors and for predicting their ultimate impact on the production environment.

Section 3.4 identifies the initial functional capabilities of a GDP requirements engineering environment (RQEE) needed to support the requirements specification and validation methodology. The ability to assimilate additional capabilities as the methodology evolves is a key feature of proposed RQEE.

Section 3.5 establishes the formal foundation for the GDP requirements specification and validation methodology. The emphasis is placed on modeling data and knowledge requirements rather than processing needs. A subset of first order logic is proposed as the principal means for constructing formalizations of the GDP requirements in a manner that is independent of the data representation. Requirements executability is achieved by selecting a subset of logic compatible with the inference mechanisms available in Prolog, a logic-based language which facilitates the expression of abstract relationships. Significant GDP concepts such as time, space, accuracy and security classification have been added to the model without losing Prolog implementability or separation of concerns. The rules of reasoning about time, space, accuracy and security may be compactly stated in second order predicate calculus and may be easily modified to meet the particular needs of a specific application. Multiple views of the data and knowledge may coexist in the same formalization.

Appendix B summarizes the results of a case study used to evaluate the formal foundation of the methodology. The formalism described in Section 3.5 is applied to used to describe the functional requirements of a hypothetical GDP system called MAPX. In the case study, the geographic information needed by MAPX (ocean depth, elevation, and rivers) is supplied in digital form. MAPX maintains two types of information: specific information about individual areas and features, and knowledge of the world in general. Specific information may have spatial, temporal, accuracy and security qualifications. World knowledge consists of constraints which must be satisfied by information maintained by MAPX, and rules used to derive new information from what is already known. The products generated by MAPX are used to illustrate both present and future requirements for output from GDP systems. The case study demonstrates the feasibility of expressing GDP functional requirements in logic and identifies some of the key technical difficulties involved in the design of the GDP requirements engineering environment.

3.2. REQUIREMENTS ENGINEERING OVERVIEW

Recent years have been marked by an increased interest in the area of system requirements specification. This is due to the realization that, in the absence of an accurate statement of purpose, the designer may solve the wrong problem, a state of affairs which often leads to disastrous consequences for all parties involved. The economic realities of system development are such that discrepancies between the delivered system and the needs it must fulfill may cost in excess of 100 times what would have been required if the errors were discovered during the initial problem definition and, in some extreme cases, they may even render the entire system useless [BOEH81].

In the simplest terms, the distinction between requirements and design specifications may be reduced to the difference between stating WHAT functions must be provided by some component and stating HOW these functions must be carried out. One immediate consequence of this fact is that requirements specifications must facilitate ease of understanding while design specifications must enable faithful rendering of physical and logical structures needed to realize the requirements.

The use of a component's design as an implicit statement of its requirements is generally not a good practice. Even for simple and well understood functions (e.g., sorting), the design may turn out to be exceedingly complex when demanding design constraints are applied (e.g., sorting in linear time). The ensuing loss of requirements traceability and separability may cause serious maintenance problems—one cannot tell if a particular function is required or is a consequence of some design constraint which is no longer significant.

The purpose of this section is to provide an overview of the current state of the art in the system requirements area in general before considering geographic data processing in particular. The presentation, based on a recent Washington University technical report, starts with a brief review of requirements specification contents and concerns. It is followed by a discussion of the formal foundation and development method of the most common requirements specification techniques. The discussion provides a backdrop against which several important requirements specification issues are being highlighted. The way these issues are being addressed by our methodology is considered in subsequent sections.

3.2.1. Requirements Specification Contents

The requirements specification serves as the acceptability criteria against which any proposed realization of some component is judged. This section reviews the kinds of information that ought to be included in a requirements specification document independent of the nature of the component for which the requirements are written. The "component" may be a whole system, a software package or a hardware device.

As shown by Yeh [YEH82], among others, requirements fall into two general categories: functional and non-functional. (The latter are also called constraints.) The functional requirements capture the nature of the interaction between the component and its environment. The non-functional requirements restrict the domain of acceptable realizations by placing constraints on the types of solutions one might consider.

3.2.1.1. Functional requirements

The construction of the functional requirements involves modelling the relevant internal states and behavior of both the component and its environment. Balzer and Goldman [BALZ79] have noted that the model, often called a *conceptual model,* must be cognitive in nature, i.e., it must involve concepts relevant to the milieu in which the component is used and should not include concepts related to its design or implementation.

The conceptual model is incomplete unless the environment with which the component interacts is also modeled. There are several reasons for this. First, if the environment is not well understood it is unlikely for the requirements, as specified, to reflect the actual needs the component must fulfill. Second, when the boundary between the component and its environment is flexible, the component complexity may be reduced by imposing certain constraints on the way the environment is expected to behave. A user, for instance, may be required to periodicly save the file being edited although the editor could be designed to be foolproof—convenience is traded for improved performance or reduced development cost. Third, the behavior of the environment is a significant factor affecting the complexity of the component design. A real-time system, for instance, is significantly less complex when the environment is known to generate stimuli at precise points in time as compared with the case when the interval between succesive stimuli is arbitrary.

Aside from defining the functional validation criterion for the component design, the conceptual model is also an important vehicle for communication between designers and users, in the problem definition stage, and between designers, during the remainder of the development life-cycle. The importance of the conceptual model increases with the complexity of the task, the risk factors associated with the project and the number of people involved.

3.2.1.2. Non-functional requirements

The degree of complexity associated with a particular design is also determined by the nature of the non-functional requirements that must be satisfied. A constraint such as high reliability, for instance, may raise significantly both the cost of the system and the level of effort associated with its design and testing.

Additional complications stem from the fact that formal specification of the non-functional requirements is difficult to accomplish. First, some constraints (e.g., response to failure) are related to possible design solutions which are not known at the time the requirements are written while others (e.g., human factors) may be determined only after complex empirical evaluations. Second, many constraints (e.g., maintainability) are not formalizable given the current state of the art. Third, not all constraints are explicit. As is the case with other engineering disciplines, a software or hardware designer is expected to follow certain generally accepted rules of the trade without having them explicitly stated. One may be justified to believe that a successful lawsuit could be brought against any company that delivers a large monolithic program on a contract. Finally, there is a great diversity of types of non-functional requirements.

A simple classification of the non-functional requirements provides sufficient evidence of their great diversity and of the difficulty involved in developing a unified and comprehensive theory on which to base some formal specification technique. We separate the non-functional requirements into six main categories:

- interface constraints
- performance constraints
- operating constraints
- life-cycle constraints
- economic constraints
- political constraints

Interface constraints deal with the precise definition of the means of interaction between the component and its environment. In the case of some application programs, the environment consists of system users, the operating system, the hardware and a software package. The functional requirements for these programs must capture the demands and services associated with each one of the environmental entities but not the syntax of the procedure invocations, the interrupt addresses or the screen format. The latter details are interface constraints that should not affect the functionality of the program.

Performance constraints are usually separated into three categories: time/space bounds, reliability and security. We add to these three survivability. The first category covers requirements concerning response time, workload, throughput, available storage space, etc. It is our expectation that user-oriented measures such as productivity will also become increasingly important in the definition of requirements for systems that provide direct production support. Reliability constraints deal with both the availability of physical components and the integrity of the information maintained or supplied by some component. Similarly, security constraints span physical considerations such as emission standards and logical issues such as permissible information flows (e.g., for secure operating systems) and information inference (e.g., from statistical summaries about the database contents). Survivability is a requirement associated not only with defense systems but also with normal data processing systems where off-site copies of the database are kept to prevent loss in case of fire.

Operating constraints include physical constraints (e.g., size, weight, power, etc.), personnel availability, skill level considerations, accessibility for maintenance, environmental conditions (e.g., temperature, radiation, etc.), spatial distribution of components, and other constraints.

Life-cycle constraints fall into two broad categories: those that pertain to qualities of the design and those that impose limitations on the development, maintenance and enhancement process. In the first group we include maintainability, enhanceability, portability, flexibility, reusability of components, expected market or production life span, upward compatibility, integration into a family of products, etc. Failure to satisfy any of these constraints may not compromise the initial delivered component but may result in increased life-cycle costs and an overall shorter life for the component. In the second group we place development time limitations, resource availability and methodological standards. The latter include design techniques, tool usage, quality assurance programs, programming standards, etc.

Economic constraints represent considerations relating to immediate and long term costs. They may be limited in scope to the component at hand (e.g., development cost) but, most often, they involve global marketing and production objectives. A high life-cycle cost may be accepted in exchange for some other tangible or intangible benefits.

Political constraints deal with policy and legal issues. A company's unwillingness to use a competitor's device and the obligation to use a certain percentage of indigenous equipment in some foreign country illustrate the type of issues falling in this category of non-functional constraints.

3.2.2. Requirements Versus Product Documentation

The product documentation is a statement of those aspects of the component that must be known by anyone wanting to use it. As such, a product document differs from a requirements specification primarily in terms of the readership it addresses. The functional requirements are identical but only a subset of the non-functional requirements may be relevant to the component user. The typical subset includes the interface, performance and operating constraints.

The similarity between the two types of specifications suggests that essentially the same techniques may be used for both. This holds the promise for significant improvements in the quality and uniformity of software product documentation. The approach is economically attractive (the requirements ought to be developed anyway) and should appease the current dissatisfaction with software product documentation which is generally characterized by incomplete functional descriptions and the absence of any performance data.

The relevance of requirements specification techniques for product documentation appears to have received little or no formal recognition in the literature, but the case when the user manual is written before the system is designed, for instance, is an implicit acknowledgment of the dual role requirements can play.

3.2.3. Important Requirements Concerns

Growing interest in the requirements specification has been accompanied by the emergence of general guidelines regarding the properties of a good specification. Our own attempt to organize and to find the motivation for these guidelines led us to adopting a functionalist viewpoint: a property of a requirements specification is desirable if it satisfies some identifiable need of the design process. This approach suggests that the way requirements are used determines the kind of properties they ought to have. Some properties are needed because the requirements must be read, others because designs must be checked against the requirements, yet others because requirements change with time during development and enhancement. Aspects that contribute to having a good requirements specification today may lose their significance in the future if the design process changes its character due to increased levels of automation or other factors.

We provide below a list of desirable requirements specification properties. The list is compiled form several sources [DUBO82, ZAVE81] and is annotated from a functionalist perspective.

Appropriateness refers to the ability of some specification to capture, in a manner which is straightforward and free of design considerations, those concepts that are germane to the component's role in the environment for which it is intended (business data processing, process control, communication hardware, etc.) Its absence may render the generation of the requirements impossible or very cumbersome.

A related property is *conceptual cleanliness*. It covers notions such as simplicity, clarity, and ease of understanding. It is needed above all because people are involved in developing and using the requirements. When the requirements are generated and used by design tools alone, conceptual cleanliness is usually sacrificed for the sake of *computational efficiency.*

Constructability deals with the existence of a systematic (potentially computer-assisted) approach to formulating the requirements. This is in recognition of the fact that the mere availability of a requirements specification formalism is not sufficient to make it useful, particularly on large problems.

Both humans and tools having to examine the requirements benefit from a *structuring* that emphasizes separation of concerns and *ease of access* to frequently needed information.

Precision, lack of ambiguity, completeness and *consistency* are important because the requirements represent the criteria against which the component acceptability is judged. Lack of precision (e.g., "large main memory") is defined as the impossibility to develop a procedure for determining if some realization does or does not meet some particular requirement. Ambiguity is present whenever two or more interpretations may be attached to a particular requirement—this is different from the case when several possible realizations are equally acceptable. A requirements specification is incomplete if some relevant aspect has been left out and is inconsistent if parts of the specification contradict one another. Both completeness and consistency require the existence of criteria against which one may evaluate the specification. While some of them may be included in the semantics of the requirements specification language, others may not. This is especially difficult to accomplish when one needs to address the consistency between multiple, related specifications such as a human interface prototype and a data-flow model of the functionality.

Consistency and completeness checks, the verification of the design against requirements, and other analytic activities presuppose the *analyzability* of the requirements by mechanical or other means. The higher the degree of *formality* the more likely is that requirements may be analyzed by some mechanical means thus opening the way to the use of tools.

Testability is defined as the availability of cost effective procedures which allow one to verify if the design and/or realization of some component satisfies its functional and non-functional requirements. This property is probably the most important one and, at the same time, the most difficult one to achieve. To illustrate the complexity involved in guaranteeing testability one may want to think of the difficulties associated with program verification where the code is checked against a set of assertions stated in predicate calculus. The problem of checking a system design against the system requirements is several orders of magnitude harder to solve and is complicated further by the fact that requirements may be application-oriented.

Traceability and *executability* of the requirements are often adequate substitutes for testability. Traceability refers to the ability to cross-reference items in the requirements specification with items in the design specification. Without assuring testability, some help is thus provided to the designer in his/her effort to check that all requirements have been considered. Executability implies the possibility to construct functional simulations of the component from its requirements specification prior to starting the design or implementation. The requirements are validated by the user/designer through experimentation with the functional simulation.

Finally, in recognition of the fact that requirements are built gradually over long periods of time and continue to evolve throughout the component's life-cycle, the specifications must be *tolerant of incompleteness* and *adaptable* to changes in the nature of the needs being satisfied by the component, they must exhibit *economy of expression*, and they must be easily *modifiable.*

3.2.4. Classification Criteria

In our earlier work we proposed a set of five classification criteria which, when applied to a requirements specification technique, help one establish its position in the requirements engineering field, i.e., its domain of applicability, intrinsic limitations and basic philosophy:

- formal foundation
- scope
- level of formality
- degree of specialization
- specialization area
- development method

For the purposes of this presentation, however, we will limit ourselves to considering only the first and the last of the criteria listed above.

3.2.4.1. Formal foundation

Significant advances in the requirements field are determined by the strength of its theoretical foundation, the basis for subsequent automation. A class of components for which formal functional requirements are routinely built is represented by compilers and interpreters. Their requirements are given by the syntax and semantics of the language for which they are constructed. Standard methods are available today for the definition of both syntax and semantics [PAGA82]. Of particular interest to the broader area of requirements specification are the three types of semantic models currently in use: denotational (the meaning of a program is stated as a mathematical function), axiomatic (the meaning of a program is stated by providing the axioms and inference rules needed to prove programs correct) and operational (the meaning of the program is given by the result of executing it on an abstract machine).

These models are expected to play an increasingly important role in the field. To date, they clearly influenced the work on the specification of functional requirements for individual programs [LISK79]. For pure procedures, axiomatic specifications take the form of input/output assertions and operational specifications are represented by simple and clear algorithms that perform the same function as the intended program but ignore any performance issues. (They are not intended for use by the actual program.) Abstract objects (appearing in object oriented design) may be specified by sets of axioms relating the operations permissible over each object or, operationally, by showing how each operation uses and modifies some abstract representation of the object.

A number of successful attempts have been made in the use of programming language semantic models as the basis for new requirements specification techniques, but their full potential has not yet been established. However, a good grasp of the principles behind the semantic models for programming languages is important in any type of design activity that involves writing or interpreting requirements and more emphasis should be placed on including this kind of knowledge in the designers' training together with more traditional formal computer science. A designer documenting an Ada® package specification, for instance, could benefit to a great extent from some knowledge of how to specify abstract objects. Similarly, a designer will find that, by not understanding the operational specification concept, he will be more likely to misinterpret requirements written in a language such as RSL [ALFO79] which has an operational nature.

Efforts to develop system level requirements specifications (i.e., the component is assumed to be a computer-based system) have explored a number of alternate formal foundations. Some approaches are based on the use of finite-state machines. Others emphasize dataflow or stimulus-response paths. Attempts have also been made to model system functionality as a community of communicating processes while data-oriented modeling of the requirements has been stimulated by efforts in the database area.

The use of finite-state machines offers elegance and a great degree of analizability. RLP (Requirements Language Processor) [DAVI79], for instance, treats system processing as a mapping that takes the current system state and an incoming stimulus and produces a new system state and a response. Redundancy, incompleteness and inconsistency in the definition of the finite-state machine are related to corresponding problems in the requirements specification.

Dataflow models are among the most popular in use today. The typical dataflow model consists of processing activities and data arcs showing the flow of data between the activities. Processing is triggered by the presence of data in the input queues associated with each activity. What makes dataflow attractive is the fact that it is very well suited for modelling the structure and behavior of most human organizations. SADT [ROSS77] and PSL/PSA [TEIC77] illustrate two distinct uses of dataflow. SADT is a requirements "blueprint language" that stresses accurate communication of ideas by graphical means; while PSL/PSA stresses the use of a requirements database and automated tools for the development and analysis of dataflow type requirements.

Techniques using stimulus-response paths decompose the requirements with respect to the processing that must be carried out subsequent to the receipt of each stimulus. The approach, rooted in the needs of the real-time processing, is widely known primarily due to the development of RSL [ALFO79].

The activities identified in both dataflow and stimulus-response models may be easily simulated by using communicating concurrent processes. Formally, a process is represented by a set of states and by a state transition mapping. This view is shared by all the techniques that model requirements using communities of processes. Fundamental differences occur mostly in the manner in which communication is being defined. In PAISLey [ZAVE81] asynchronous interactions are specified by means of function applications. (So called "exchange functions" allow the passing of information via their arguments and returned values). Jackson [JACK78] uses unbounded queues defined such that only one process may "write" to each queue and only one process may "read" from each queue. IORL [TELE82] provides a set of eight communication primitives that permit the establishment of communication paths and the exchange of data.

Data-oriented techniques concentrate on the specification of the system state represented by the data that needs to be maintained. Built-in data manipulation primitives are used to construct specifications for the system functionality. CSDL [ROUS79], for instance, is a conceptual schema definition language used to structure one's knowledge of the application area. Based on semantic network modelling and other techniques proven successful in the artificial intelligence field, CSDL provides a highly abstract and intuitive representations of knowledge.

The techniques we have cited are representative of the significant progress registered during the last decade but also indicative of some of the limitations of the current state of the art. Since no technique is equally appropriate for all applications or comprehensive in its coverage of the requirements issues. significant effort should be directed toward evaluating the potential of proven techniques in new contexts. Furthermore, there is a need to expand the formal foundation of the requirements area. It is clear, for instance, that probabilistic concepts have not received appropriate attention, that logic based models are just beginning to be exploited, that axiomatic methods for dealing with the specification of behavior (event sequencing) is only a theoretical concern, etc.

3.2.4.2. Development method

Recent years brought about a new distinguishing factor among requirements specification techniques: the development method. While the prevailing approach is to state the requirements completely before proceeding with the design, rapid prototyping has made significant gains in popularity. As recent studies [BOEH84] show, both methods have advantages and disadvantages. Rapid prototyping seems to lead to less code, less effort and ease of use while the traditional approach is characterized by better coherence, more functionality, higher robustness, and ease of integration. More importantly, these results could be interpreted as suggesting the need to use a mixed approach where one uses a subset of the requirements to develop rapid prototypes which in turn lead to further clarification and refinement of the original requirements. (Because of the relation between requirements and product documentation we consider that even pure rapid prototyping does lead to a statement of requirements. It should be noted that while requirements as such may never be generated, product documentation is still needed—at least in the form of on-line user documentation.)

Two more exotic methods are also making their beginnings in the requirements field. The first one is represented by efforts to introduce expert knowledge-based systems into the process of developing the requirements. (The 7'th International Conference on Software Engineering in March 1984, for instance, included one session on this topic.) The second one relates to defining requirements for situations where the problem is extremely ill specified (e.g., a medical diagnosis system). This situation is very common in the artificial intelligence community and the usual solution is not to specify the "functionality" but an evaluation procedure and a set of related acceptance criteria (e.g., 90% agreement with some group of experts on a predefined set of cases).

Many of the shortcomings we see in the today's approaches are due to the underlying assumptions being made about how requirements ought to be developed. While current strategies tend to structure the requirements specification process, it is hoped that future requirements development stategies will provide instead a milieu for reasoning about the problem at hand. However, the systematic investigation of new requirements development strategies has just been started and its full impact remains still to be determined.

3.2.5. Concluding Remarks

It is clear that, despite significant growth, the requirements area still faces a number of important unresolved issues and suffers form a lack of crystalization. The formal foundation of the field must be broaden by evaluating the capabilities of different types of formalisms (e.g., logic, probability theory, etc.). A theoretical foundation for the specification of non-functional requirements still needs to be established. The degree of formality must be increased in order to reach greater levels of automation. The designers' abilities to deal with formality must be enhanced through proper training and new forms of automation that take into consideration the human factor and incorporate more domain specific concepts in the requirements. New methods for developing requirements specifications must be considered. A major integration effort must be undertaken for the purpose of establishing a unified formal foundation that could bring together application and design oriented specifications, functional and non-functional requirements, the life-cycle phases, and design and requirements definition activities.

During our efforts to develop a GDP requirements specification and validation methodology, we have given considerable attention to the issues noted here and we made a concerted effort to address many of them. Without solving all of them, it is our contention that we have made significant strides in several key directions.

3.2.6. References

[ALFO79] Alford, M., "Requirements for Distributed Data Processing Design," *Proc. 1'st Int. Conf. on Distributed Computing Systems*, pp. 1-14, October 1979.

[BALZ79] Balzer, R. and Goldman, N., "Principles of Good Software Specification and Their Implications for Specification Languages," *Proc. Specifications of Reliable Software Conf.*, pp. 58-67, April 1979.

[BOEH81] Boehm, B. W., *Software Engineering Economics*, Prentice-Hall, Inc., 1981.

[BOEH84] Boehm, B. W., Gray, T. E. and Seewaldt, T., "Prototyping vs. Specifying: A Multi-Project Experiment," *Proc. of 7'th Int'l Conf. on Soft. Eng.*, pp. 473-484, March 1984.

[DAVI79] Davis, A. M. and Rauscher, T. G., "Formal Techniques and Automatic Processing to Ensure Correctness in Requirements Specifications," *Proc. Specifications of Reliable Software Conf.*, pp. 15-35, April 1979.

[DUBO82] Dubois, E., Finance, J. P. and Van Lamsweerde, A., "Towards a Deductive Approach to Information System Specification and Design," *International Symposium on Current Issues of Requirements* Engineering Environments, Y. Ohno editor, pp. 23-31, OHM/North-Holland Pub. Co., 1982.

[JACK78] Jackson, M. A., "Information Systems: Modelling, Sequencing and Transformations," *Proc. of 3'th Int'l Conf. on Soft. Eng.*, pp. 72-81, May 1978.

[LISK79] Liskov, B., and Berzins, V., "An Appraisal of Program Specifications," *Research Directions in Software Technology,* P. Wegner, editor, MIT Press, pp. 276-301, 1979.

[PAGA82] Pagan, F. G., *Formal Specification of Programming Languages: A Panoramic Primer,* Prentice-Hall, Inc., 1982.

[ROMA84] Roman, G.-C., Stucki, M. J., Ball, W. E. and Gillett, W. D., "A Total System Design Framework," *Computer* 17, No.5, pp. 15-26, May 1984.

[ROSS77] Ross, D. T., "Structured Analysis (SA): A Language fo. Communicating Ideas," *IEEE Trans. on Soft. Eng.* SE-3, No. 1, pp. 16-34, January 1977.

[ROUS79] Roussopoulos, N., "CSDL: A Conceptual Schema Definition Language for the Design of Data Base Applications," *IEEE Trans. on Soft. Eng.* SE-5, No. 5, pp. 481-496, September 1979.

[SCHE84] Scheffer, P. A., Stone, III, A. H. and Rzepca, W. E., "A Large System Evaluation of SREM," *Proc. of 7'th Int'l Conf. on Soft. Eng.*, pp. 172-180, March 1984.

[TELE82] "IORL Version 2 Language Reference Manual," Teledyne Brown Engineering, December 1982.

[TEIC77] Teichroew, D. and Hershey, III, E. A., "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," *IEEE Trans. on Soft. Eng.* SE-3, No. 1, pp. 41-48, January 1977.

[TSIC82] Tsichritzis, D. C. and Lochovsky, F. H., *Data Models,* Prentice-Hall, Inc. 1982.

[YEH82] Yeh, R. T., "Requirements Analysis—A Management Perspective," *Proc. of COMPSAC'82,* pp. 410-416, November 1982.

[ZAVE81] Zave, P. and Yeh, R. T., "Executable Requirements for Embedded Systems," *Proc. of 5'th Int. Conf. on Soft. Eng.*, pp. 295-304, March 1981.

3.3. REQUIREMENTS SPECIFICATION AND VALIDATION

Good understanding and proper formulation of the system requirements are the cornerstone for the development of a system that is responsive to the evolving user needs. Reaching the required understanding, however, is a very difficult task supported only to a limited extent by currently available requirements specification techniques. Although they add structure and precision to the specification process, they do not create a milieu for reasoning, visualizing, exercising and evaluating the requirements and, to a large extent, they isolate requirements from the larger context of the system life-cycle. In other words, they treat requirements development and evaluation primarily as a specification process rather than a discovery and thought process. While the technical difficulties of assuming the latter point of view are quite obvious, by limiting the domain of discourse to a somewhat narrower area of interest, such as geographic data processing (GDP), there is the opportunity for significant progress toward a more cognitive approach to requirements specification.

Our objective is to develop a GDP requirements specification and validation methodology (RSVM) which

(1) is production oriented;

(2) covers a broad range of requirements concerns including data and knowledge requirements as well as security and accuracy requirements;

(3) addresses the clarification and evaluation of non-functional requirements such as human factors, and performance;

(4) ties requirements to questions about technological trends and production objectives of the user organization;

(5) integrates requirements and design evaluation thus maintaining high visibility and relevance of the requirements throughout the design process and enabling one to assess early the adequacy of proposed technical solutions (e.g., systems, algorithms) to problems arising from current or postulated production needs and the ultimate impact these technical solutions will have on the production environment;

(6) provides the means for logical reasoning about requirements; and

(7) includes the means for visualizing both requirements and conclusions drawn from reasoning about them.

Rooted in the notion of rapid prototyping, RSVM's basic strategy is to build simulators for the functionality of proposed systems, or parts thereof, and to record intermediary results of the simulation for the purpose of an interactive, real time playback of the simulation script. Built-in adjustable delays reproduce the performance characteristics of the technical solution under consideration. In this way both the functional adequacy and the potential production impact may be clearly identified prior to committing to the actual purchase or development of the system. The evaluators may be either technical experts or actual production personnel and the simulations may be used both as a means for technical evaluation and as an avenue for building and evaluating training materials for new production systems. Reasoning capabilities are provided by a logic-based underlying formal model of GDP requirements while the visualization is achieved by means of graphic rendering of logical information.

This section provides an outline for the methodology (Figure 3.3-1). The outline should be viewed as the starting point for the methodology development and assessment. Successive refinements will have to be generated by applying it in the context of various realistic case studies. With each iteration, the methodology is expected to grow in scope and sophistication while the use of the simulations will help us better understand how to make them more meaningful and how to use them in the evaluation of the long range impact of highly integrated data/knowledge bases. Following the methodology outline we will discuss the nature of the production process model in preparation for the description of its formal foundation in a later section.

3.3.1. Methodology Overview

Selection of the production process to be evaluated.

The application of the methodology starts with selection of the production process to be evaluated. This step assumes different forms depending on the nature of the investigation. In some cases the production process may be the one supported by some new system whose design is being evaluated. In other cases some aspect of the existing production processes may be selected as a candidate for improvement, streamlining or exploration.

Development of a preliminary production process model.

The development of a preliminary production process model starts with a task analysis of the production activities and with the establishment of the role played by the cartographer. (The term cartographer is used to mean any DMA personnel involved in the production process.) The purpose of this step is to develop a preliminary statement of requirements for the system under consideration. The key issues to be addressed are the definition of the behavior exibited by the system during its interaction with the cartographer and the identification of the data and operations needed to continue the model construction.

The level of detail to which the behavior must be specified depends upon its effect on performance and functionality. The level of abstraction must be high enough as to avoid unnecessary complexity but also low enough as to be able to determine any important effects on the functionality and peformance characteristics of the system. A finite state graph is probably satisfactory for many GDP systems. The simulation command language (SICOL) provided by the requirements engineering environment supporting the methodology offers the means by which a tentative simulation script embodying the system behavior may be constructed. Later steps will have to further refine the bahavior by substituting more detailed behavior specifications for each node in the graph.

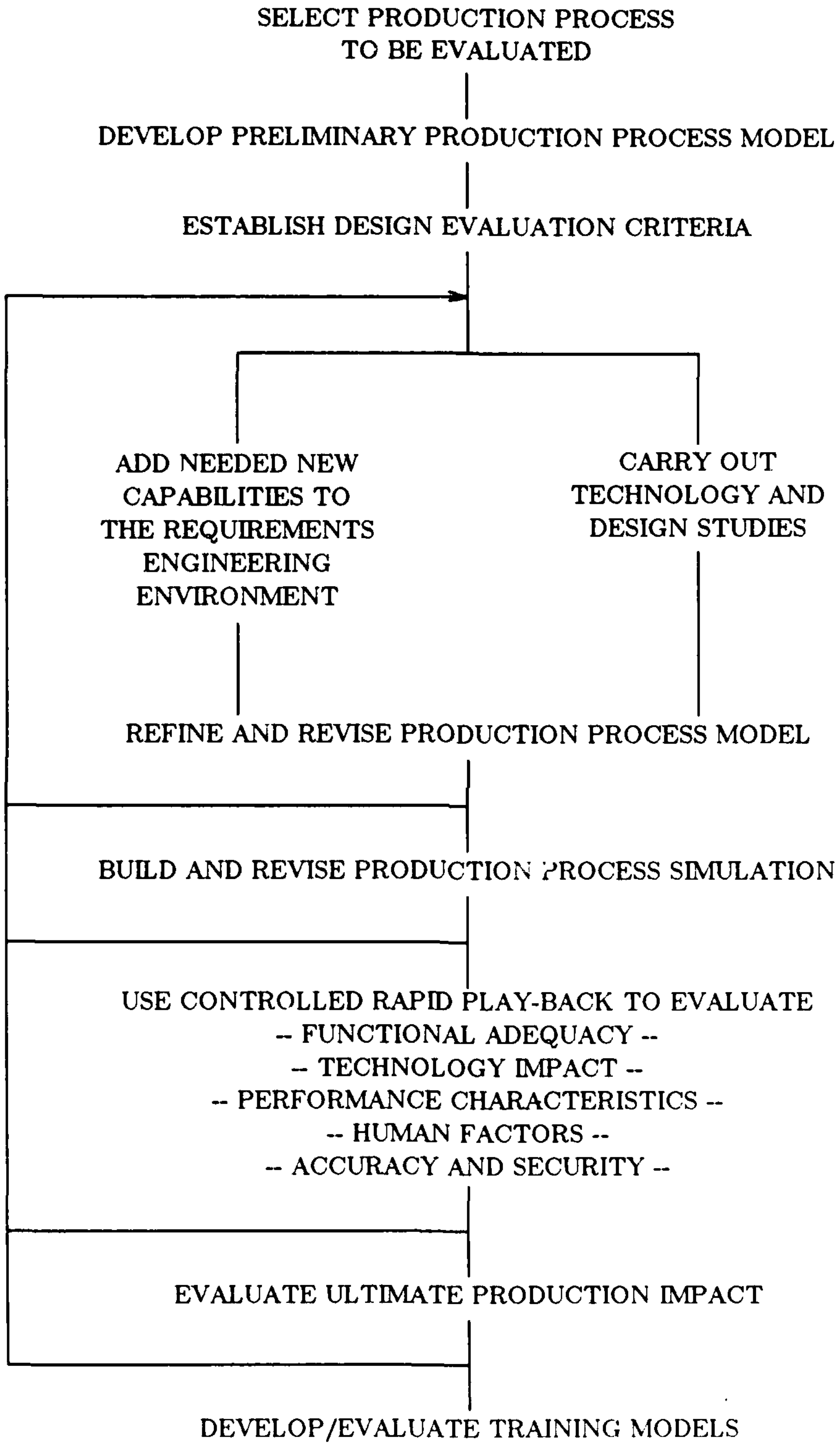

SELECT PRODUCTION PROCESS TO BE EVALUATED
DEVELOP PRELIMINARY PRODUCTION PROCESS MODEL
ESTABLISH DESIGN EVALUATION CRITERIA
ADD NEEDED NEW CAPABILITIES TO THE REQUIREMENTS ENGINEERING ENVIRONMENT
CARRY OUT TECHNOLOGY AND DESIGN STUDIES
REFINE AND REVISE PRODUCTION PROCESS MODEL
BUILD AND REVISE PRODUCTION PROCESS SIMULATION
USE CONTROLLED RAPID PLAY-BACK TO EVALUATE
-- FUNCTIONAL ADEQUACY --
-- TECHNOLOGY IMPACT --
-- PERFORMANCE CHARACTERISTICS --
-- HUMAN FACTORS --
-- ACCURACY AND SECURITY --
EVALUATE ULTIMATE PRODUCTION IMPACT
DEVELOP/EVALUATE TRAINING MODELS

Figure 3.3-1: THE GDP REQUIREMENTS SPECIFICATION/VALIDATION METHODOLOGY

A byproduct of the behavior specification is the identification of certain data and operational requirements. Their actual specification, however, remains to be carried out by later steps.

Establishment of design evaluation criteria.

The understanding gained during the development of the preliminary production process model must also contribute to the establishment of the evaluation criteria which will be applied to the system under consideration. Since the GDP system is not an end unto itself but a production tool the emphasis should be placed on adopting a production oriented viewpoint. Any attempts to translate production objectives into technical objectives is counter-productive in the very early stages and incurs the risk of misstating the organization's real objectives. Furthermore, because these criteria affect the complexity of the simulation, the nature of the analytical and predictive studies as well as the related technical investigations, great care must be taken to validate them with respect to the ultimate objectives of the organization. Priorities must be assigned and general trade-off guidelines should be drafted.

Addition of needed new functional capabilities to the requirements engineering environment.

The functional capabilities of the requirements engineering environment (RQEE) are expected to grow because each new study may need specific functional capabilities. However, once the new capabilities are integrated into the requirements engineering environment they become available for use in future endeavors. This approach allows the facility to evolve in response to the changing needs of DMA production and avoids the requirement for an initial high risk, high investment effort in establishing the RQEE.

The need for new functional capabilities arises naturally from the fact that the RQEE is anticipated to serve investigative efforts that vary in scope and method. The way different functional needs are met also varies from one study to another: some new image processing algorithm may have to be developed or customized, a collection of feature extraction algorithms may have to be gathered, primitives enabling the manual simulation of some difficult to implement production process may have to be designed, some new graphic rendering of stored information may have to be conceived, etc. In each case, once acquired, the new capability becomes part of one of the requirements engineering environment and may be referenced or invoked through the simulation command language.

Technology and design studies.

Technically meaningful simulations must extend beyond functional simulation into rendering the performance characteristics of postulated designs. This requires one to carry out technology and design studies to determine the technological options and system architectures that one should to consider during the investigation (unless the design is a given) and to establish the performance characteristics of the various functions being simulated. We envision extensive use of both analytical and discrete event simulation methods as part of these studies. The time taken by an image data base retrieval, for instance, could be determined analytically and later included as a delay during the simulation play-back.

An important concern of these studies is the identification of relevant technological assumptions, i.e., the formalization of the relation between the key design parameters (e.g., number of parallel processors) and the simulation features they control (e.g., behavior, time delays).

Production process model refinement and revision.

This information together with the knowledge regarding the added functional capabilities is later used to refine and revise the production process model. In turn, this step may identify new capabilities and design studies to be produced by the previous steps which play a support role.

We see the requirements specification to be a highly iterative activity that starts with the development of the overall system behavior specification and grows as the definition of its behavior, data and operations is being refined and revised. Initially, refinements dominate as the designer attempts to establish the desired functionality of the system. As the functional requirements converge toward a stable specification the design studies start being initiated. At this point revisions become dominant as attempts are being made to aleviate newly discovered performance shortcomings. The order in which refinements and revisions should occure depends on the nature of the problem at hand, but some general guidelines are provided in the next section where the production process model is discussed in more detail.

Production process simulation building and revising.

The building of the production process simulation involves putting together: the functional requirements specifications; the models of the system/cartographer interactions; the definition of the play-back controls including the default values for the built-in adjustable delays; and the means for recording intermediary results which will later permit the rapid play-back of the simulation. The work is guided by the definition of the simulation script built in a previous step, but continued refinements to the script are also expected.

System evaluation using controlled rapid play-back.

Upon completion of the simulation, controlled rapid play-back is used to reproduce the production process as seen by the cartographer. The simulation play-back allows one to analyze issues falling into four categories: functional adequacy, technology impact, performance characteristics and human factors.

The concept of functional adequacy is broader than functional correctness. A program designed to add two integers is correct if, for any two values it adds, the expected result is the sum of the two values but it may be functionally inadequate if it accepts as input only values less than one hundred. Geographic data processing often involves functions that are not as well defined as the addition operation, e.g., cultural feature extraction. In such cases, different algorithms designed to solve the same problem exibit varying degrees of adequacy depending on the domain of discourse and the degree of human intervention that is permissible.

Functional adequacy is valuable for both requirements and design validation. In one case it is used as an aid in determining the functional capabilities of a system prior to starting the design effort; in the other case, a system design may be checked to see if it supports the needed functionality. Functional adequacy may be established formally or empiricly. If the results of the production process are rigorously specified, a formal evaluation may take the functional model and prove logically its ability to generate all desired results. This possibility exists because of the logic foundation on which the functional capabilities provided by the RQEE are built.

The empirical evaluation may take place during the building of the simulation or during play-back. The interactive nature of the simulation combined with the use of cartographers and other personnel familiar with the production problem enhances the accuracy of the evaluation. Moreover, the input data may be manipulated in ways that allow the cartographer to carry out what we choose to call "a stress analysis of the algorithms." This involves the development of customized input data designed to test the point at which the algorithm becomes inadequate and to determine how other algorithms might handle the same data.

The impact of the underlying technology is factored into the simulation primarily in terms of the built-in delays. By permitting changes in the values of the delays one may observe the anticipated effects of corresponding technological changes. Since the evaluator may be a different person from the developer of the simulation, he/she does not specify delay values but the selection of a particular technological alternative which results in the appropriate adjustments of the delay values. The same approach is used to evaluate the effects of changes in the system load and other factors affecting performance.

The evaluation of human factors is carried out using well established empirical methods. The results of the human factors studies feed into interface steamlining efforts, the establishment of productivity estimations, and the development of training materials for the production personnel.

Evaluation of the ultimate production impact.

The productivity estimates together with the simulation script (which is a model of the production process) may be used to evaluate the ultimate production impact of the system under consideration. This will help the organization predict more accurately the effect of planned changes to the production process. This approach is in keeping with the general philosophy behind the methodology which subordinates the technical considerations to production objectives and attempts to bridge the gap that often exists between the two. The ultimate result is that systems may be evaluted by the people that need them rather than indirectly through technical experts that may have only a limited understanding of the intricacies of a production oriented organization. Moreover, the opportunity is created for global production oriented optimization.

Development of training materials.

Because the simulations may be easily transferred to an optical disk, it is possible to use them as the basis for the development of training materials. In principle this requires the additional of some tutorial component to the original simulation script. At this time, it appears that no new capabilities are required to accomplish this additional objective. The use of the optical disks in dedicated stand-alone training stations also appears to be a feasible option to be considered at a later date.

3.3.2. The Production Process Model

The methodology we propose constitutes a highly effective approach to the specification and validation of GDP requirements only when supported by an appropriate requirements engineering environment (RQEE). This section discusses in general terms the capabilities needed from the RQEE and how they influenced the nature of the production process model. As shown in Figure 3.3-2, the RQEE capabilities fall into five major categories:

SPECIFICATION capability
which is the ability to state the system's requirements;

VISUALIZATION capability
which is the ability to generate graphic renderings of the information maintained by the RQEE;

REASONING capability
which is the ability to infere new information from the information maintained by the RQEE;

EXERCISING capability
which is the ability to produce functional simulations of the requirements, i.e., to achieve what is commonly called requirements executability.

EVALUATION capability
which is the ability to predict various qualitative and quantitative system characteristics using the requirements specifications and the design solutions being proposed.

We turn next to considering the demands these capabilities place on the nature of the production process model.

3.3.2.1. Specification capabilities

The requirements specification of a GDP system is represented by: the behavior the system exibits while interacting with the cartographer; the information it maintains (which in turn may be classified as data and knowledge); and the operations, (i.e., information processing algorithms) the system makes available to the cartographer.

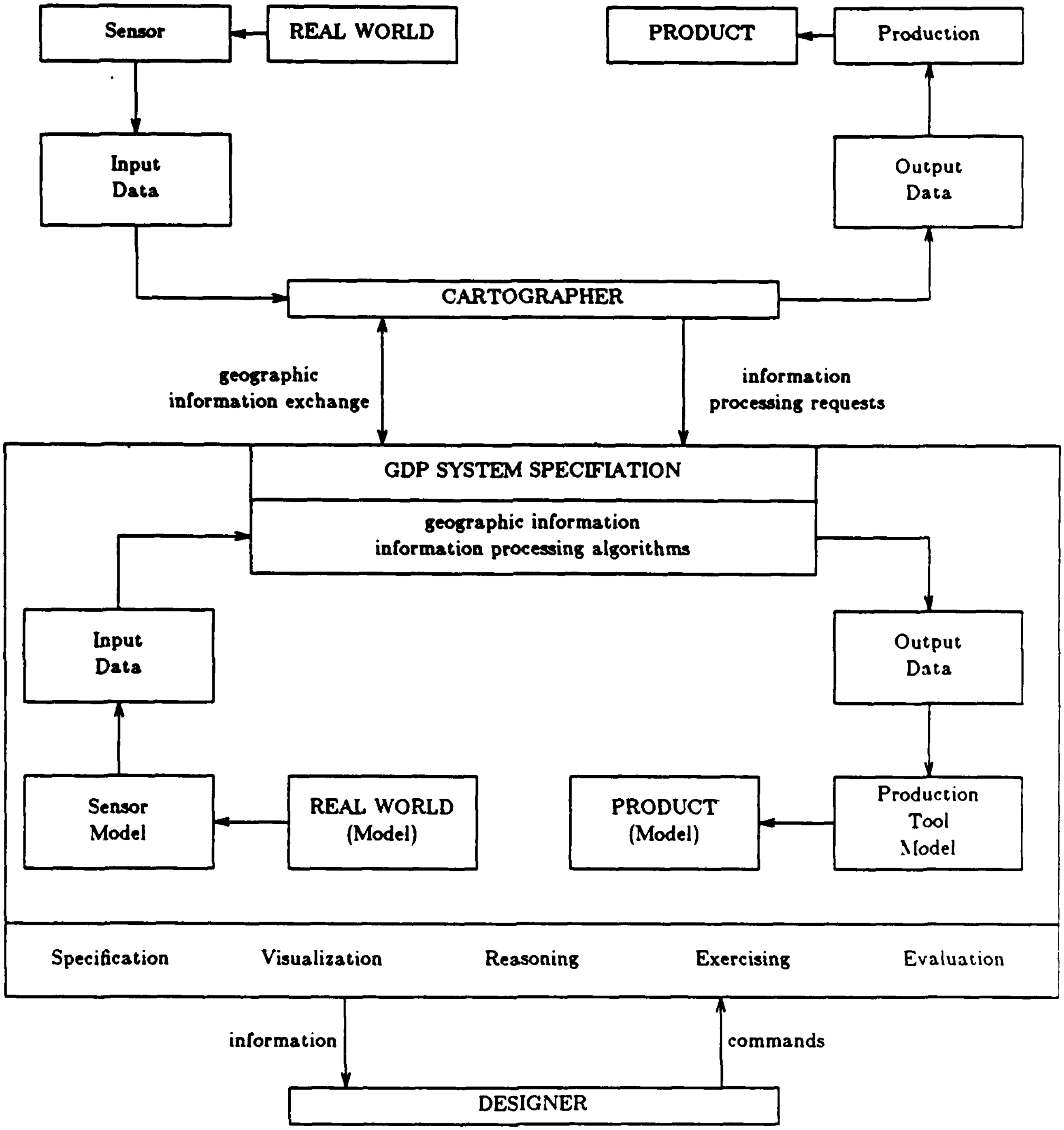


Figure 3.3-2: THE REQUIREMENTS ENGINEERING PERSPECTIVE

Behavior.

The behavior is defined as the sequence of actions the system allows the cartographer to initiate. At the highest level of abstraction, the behavior describes the overall production flow by identifying the major states of the system and the activities that trigger the transition from one state to the next. It is our contention that a finite state graph suffices to specify this highly abstract definition of the system's behavior. This is the starting point for the behavior specification. Succesive refinements, however, should eventually lead to constructing precise definitions of the human interfaces needed to build a meaningful system prototypes.

The development of the complete behavior specification is seen as a two-phase process each proceeding more or less top-down. During the first phase, a high level abstraction of the system's behavior, called an abstract interface, is defined in terms of a finite state graph. Nodes in the graph are later replaced by subgraphs representing further refinements of the system's behavior. At some point, however, additional refinements will have to capture the actual details of the the human interface. This marks the start of the second phase during which programs that simulate the human interfaces are being built and attached to the nodes of the abstract interface. Easy development of such programs could be facilitated by the use of specialized tools. During simulations these programs are invoked whenever the corresponding node is reached.

Data Specification.

GDP systems are, above all, repositories of geographic data, which consists of known facts about various geographic entities. Some facts are independent of any particular location but most are true only at specific positions on the globe defined by the respecitve latitude and longitude or equivalent. Many facts are further qualified with respect to the time of the recording, their believed accuracy and their assigned security. Furhermore, since most facts are relative with some point of view or useful only for some GDP products they may be categorized and assigned to various classes of facts we choose to call models. Model membership may be viewed as yet another qualification of facts.

Because data specification ought to be free of implementation considerations during requirements definition, we selected logic as the means of data specification. Sample facts used to test the requirements may be specified using a subset of first order predicate calculus. The approach, described in detail in a subsequent section, is also the key to enabling the designer to reason about requirements.

The development of data specifications for representative data samples occurs concurrently with the refinement of the system's behavior and must start with the defintion of the input data needed to build the system's internal data. While output data specification may be postponed until product development becomes an important concern, the system data must be specified as early as possible since it is a key component of the system state and must be reconcilied (for consistency purposes) with the states identified in the abstract interface.

Knowledge Specification.

Restrictions on the nature of individual or groups of facts and general rules of reasoning are the most common forms of knowledge we see being specified in the context of a GDP system. Although most current GDP systems do not involve explicit knowledge representation, efforts are under way to exploit existing artificial intelligence techniques for the benefit of geographic data processing. Moreover, it is our belief that by making explicit the system's view of the real world the number of specification errors may be reduced by a significant degree while enhancing the ability to correctly evaluate the potential impact of proposed changes in the specifications. The knowledge specifications may be used by the designer to understand the nature of the problem at hand, to check data integrity, to devise artificial intelligence techniques for solving specific problems, etc.

Because general knowledge is usually independent of particular facts, the use of second order predicate calculus appears to be the most obvious path to follow. In our studies we selected and used a subset of second order predicate calculus to define rules for reasoning about space, time, accuracy and security to formulate a variety of data integrity rules.

Knowledge specifications evolve along with the system data specifications as the designer achieves an increasingly better understanding of the application. Building and reasoning about specific representative data samples often speed up this process.

Operation Specifications.

Operations are performed on data to generate new data. Their complexity may vary greatly from traditional mathematical manipulation of data, to image processing, to logical manipulations and, finally, to complicated expert systems.

Typically, the operations are identified during the development of the behavior specifications because they are carried out in direct response to a cartographer's request—it is the cartographer who, at least for the time being, controls the production process. If possible, non-procedural specifications should be built as soon as the operation is identified using the data and knowledge specifications. Eventually, due to the need to develop the rapid prototype, operational specifications or even the actual algorithms will have to be developed. During simulations the operations are invoked at predefined points in the behavior specification. We anticipate that, quite often, the need to evaluate a prototype for a particular point of view for which the accuracy of some operation is not essential will result in the use of specialized high performance versions of some of the algorithms. Such versions would execute correctly only for the data used in that particular simulation. Moreover, when the algorithms remain to be developed at a later date, manual simulations of the algorithm's effects will have to be supplied to the simulation.

3.3.2.2. Visualization capabilities

In order to significantly enhance the designer's capability to acquire and analyze large volumes of information about the system's requirements, powerful visual representations of the data must be made available on demand. The mapping from logical facts to images could be done directly. Additional power and flexibility is gained, however, if one introduces the concept of a visualization domain and breaks the transformation into two. First, a mapping from logical facts to an abstract representation of some imaginary two- or three-dimensional object that appeals to the designer's intuition, e.g., histogram in the shape of the U.S. map, 3D-surface, etc. Second, a mapping of the abstract object (corresponding to a mental image) to visual images that may be interactively manipulated, modified and examined.

A 3D-surface, for instance, could be viewed from different angles, using different light sources, using different color assignments, in cross-section, etc. A 2D-image may be filtered to enhance the ease with which certain types of faults may be detected through visual inspection and may be compared with images from which the logical facts have been extracted.

Because all information is represented using logic, by providing the mechanisms for data visualization, any other information derivable by the designer by means of logical inference may also be rendered visually in the same manner. Specialized information rendering techniques are expected to evolve as the RQEE is first used, with the techniques already in use today being the first candidates for incorporation in the RQEE.

3.3.2.3. Reasoning capabilities

The basic ability to carry out logical inference comes with the logic based foundation of the data and knowledge specification. Simple logic expressions may be formulated for the purpose of defining arbitrary data extraction criteria, to investigate relationships beetwen facts, to establish logical consequences of known facts, to reveal potential contradictions, etc. Moreover, application-specific rules of reasoning may be developed using second order predicate calculus and their use may be limited only to those circumstances where they are applicable. The ability to define such rules is essential if application oriented concepts are to be incorporated into the requirements specification and also in order to overcome the problems related to dealing with general concepts for which no adequate general logical theory exists. Time, space, accuracy and security all fall in the latter group.

Reasoning about requirements may be taken even further. First, when certain non-functional requirements, e.g., accuracy and security, may be formally specified in logic, the system's ability to satisfy them may checked mechanically or, sometimes, proven formally. Second, it is possible to allow the designer to reason about the relation between the system's data and the real world with which the system is concerned. This may be done if, in addition to the aspects traditionally included in the system's functional specification, the production process model includes accurate and detailed models of some area of the real world, sensors, production tools and products along with representation independent specifications of the input and output data. (See Figure 3.3-2.)

These models may be used in the logical verification and accuracy evaluation of the requirements specification. Input test data may be generated from the real world model and, after being processed by a prototype version of the system, the resulting product may be checked against the expected product generated independently (manually or by some existing system). Furthermore, the information maintained by the system may be compared against that included in the real world model thus providing an objective, less error prone, and mechanized verification procedure.

This approach is feasible due to two important reasons: the type of application area under consideration and the nature of the requirements engineering environment we are envisioning. Geographic data processing makes it feasible to construct a highly accurate model of a typical yet small subset of the world about which a system maintains information. Because these models may be specified using the same logic-based formalization intended for the system specifications, the designer may use identical mechanisms to examine, reason about and visualize information regarding the real world, input data, output data and products as well as the information maintained by the system.

3.3.2.4. Requirements exercising capabilities

One point of difficulty in the verification of functional requirements is the size and complexity of the specifications that must be verified. The problem is aleviated to some extent if the requirements are executable, i.e., if functional simulations of the system may be generated directly from the requirements. The designer evaluates the requirements not only by visual inspection but also by submitting sample input data and examining the results that would be generated by the specified system. RSVM extensively exploits requirements executability in the construction of rapid prototypes that extend to the point of including sensor and production tool models.

We achieved requirements executability by emphasizing the use of operational specifications for the processing algorithms and by ensuring that the logic based data and knowledge specifications are fully implementable in an existing logic based programming language, Prolog. A simple mechanical translation can reformulate in terms of Prolog all first and second order predicate calculus specifications we proposed, despite the fact that Prolog is designed to handle only a subset of first order predicate calculus. Thus the powerful logical inference mechanisms of Prolog become available to support the needed reasoning capabilities. Moreover, our studies have revealed that some of the performance difficulties relating to Prolog may be overcome by designing appropriate interfaces to a more efficient language such as C and to existing image processing software.

3.3.2.5. System evaluation capabilites

RSVM is concerned with the validation of both functional and non-functional requirements. The functional requirements include the system behavior being perceived by the cartographer, the system state determined by the geographic information it maintains and the possible state transitions which are defined by the information processing algorithms the system makes available to the cartographer. The non-functional requirements are represented by a set of criteria that must be met by the system in order to provide adequate support to the production activities of the organization. The types of acceptance criteria being considered by RSVM fall into the following categories: functional adequacy, technology utilization, performance, human factors, security, accuracy, and production impact.

Functional adequacy and human factors are investigated with the aid of sophisticated rapid prototyping techniques. Technology utilization and performance are studied using traditional appraches to system design evaluation with the results being fed into the functional simulations in the form of processing delays. In this manner the evaluation of all the technical concerns mentioned so far may be integrated and trade-offs among them may be identified more clearly. The production impact may be derived accurately by exercising and performing measurements on the prototypes. Finally, security and accuracy requirements may be checked by comparing the properties of the system data against independently stated security and accuracy requirements.

3.3.3. Concluding Remarks

The primary use of RSVM is in the area of geographic data processing. Within this context, however, it exceeds the scope of the traditional requirements definition methodologies. This is because requiremets are viewed from the perspective of the entire system life-cycle: from problem definition to integration into the actual production. RSVM may be used

(1) to define the user needs and the way they change;

(2) to evaluate specific design proposals and potential enhancements;

(3) to investigate the impact of technological trends;

(4) to determine the computer system's ultimate impact on production; and

(5) to develop appropriate training materials and programs for the use of new or existing systems.

This unprecedented broadness of scope is made possible by integrating a number of proven development, evaluation and planning techniques. They include rapid prototyping, functional simulation, performance evaluation, production planning, and human factors studies.

3.4. REQUIREMENTS ENGINEERING ENVIRONMENT

The proposed RSVM postulates the existance of a requirements engineering environment (RQEE) that provides certain basic functional capabilities needed in order to apply the methodology. It is the objective of this section to identify these capabilities and the hardware and software resources needed to supply them.

3.4.1. Functional Capabilities

A distinctive feature of our methodology is the fact that it goes beyond the mere specification and validation of functional requirements by addressing issues concerned with the clarification of certain quantitative and qualitative aspects of selected non-functional requirements. This is accomplished by using performance data generated by system design studies in the refinement and validation of the requirements. This makes the RQEE also useful in the evaluation of both general technological trends and specific design proposals with respect to selected DMA production objectives, particularly when high risks and costs may be involved. The ability of a design to achieve the stated objectives is determined through a coordinated assessment of human and technical factors (e.g., functional adequacy, technology impact, performance).

Because RSVM accomplishes this coordinated assessment by means of realistic and technically meaningful simulations developed by technical experts but "tested" using actual DMA production personnel, RQEE will have to provide appropriate interfaces to three types of people:

(1) CARTOGRAPHERS—evaluators selected by DMA to participate in the human factors part of the simulation assessment;

(2) DESIGNERS—technical experts that build simulations and carry out technology and design studies using RQEE;

(3) DEVELOPERS—technical experts responsible for the upgrading of the RQEE's functional capabilities.

It should be noted that the distinction between designers and developers is a functional one since we anticipate that the growth in functional capabilities will be designer directed, i.e., triggered by the specific needs of technical studies using the facility. In this manner smooth, responsive and low risk evolution of the RQEE will be assured long into the future.

The interface between the cartographer and RQEE is specific for each simulation. In other words, the cartographer does not "see" RQEE but some production process that might be acquired by DMA. This will require the designer to be able to define in the simulation the human interface paradigm used by the particular design under study.

The designer perceives RQEE as a simulation building and evaluation facility consisting of an integrated collection of resources (Figure 3.4-1) accessible through a single interface: a *simulation command language* (SIMCOL). SIMCOL's commands fall into four categories:

SCRIPT DEFINITION

The simulation script is a model of the production process being simulated. Each script is stored in the symbolic database and may be simply viewed as a finite state graph (Figure 3.4-2) where the nodes, called *frames,* correspond to significant intermediary results of the production process and the transitions represent the flow from one intermediary result to the next. The script definition commands are used to create, modify and destroy scripts.

SIMULATION GENERATION

The script is used to guide the simulation generation. The latter involves the definition of the simulation environment (e.g., world models being used, play-back modes, etc.) and the attaching, to frames, of processes that simulate subparts of the production process or invoke precomputed intermediary results. The intermediary results (images or abstractions) are created and stored by accessing the various databases and by using algorithms available in the libraries.

The processes associated with each frame are described by providing:

An image representing some intermediary result in the production process that must be displayed when the frame is reached. The image may exist already in the image database at generation time, may be built by the designer and placed in the image database as part of the simulation generation process or may be built during the simulation.

The definition of the interactions between the cartographer and the simulation during the rapid play-back of the simulation. They may include image transformations using preselected algorithms from the libraries.

The definition of delays to be used during rapid play-back in order to simulate the performance profile of the design under consideration.

The rules by which the next frame is selected in response to the the actions taken by the cartographer.

PLAY-BACK CONTROLS

Rapid play-back of the simulation is made possible by the storing of intermediary results which are referenced in the frame definition and may be recalled immediately whenever some frame is reached. The play-back may be invoked at the start of an evaluation involving a cartographer or during the simulation generation in order to assist the designer with the debugging or evaluation of portions of the simulation. In the former case, only very simple controls are required beyond those integral to the simulated cartographer/system interface (e.g., start, pause, continue, stop and restart). In the latter case, the designer needs significantly more control: dynamic

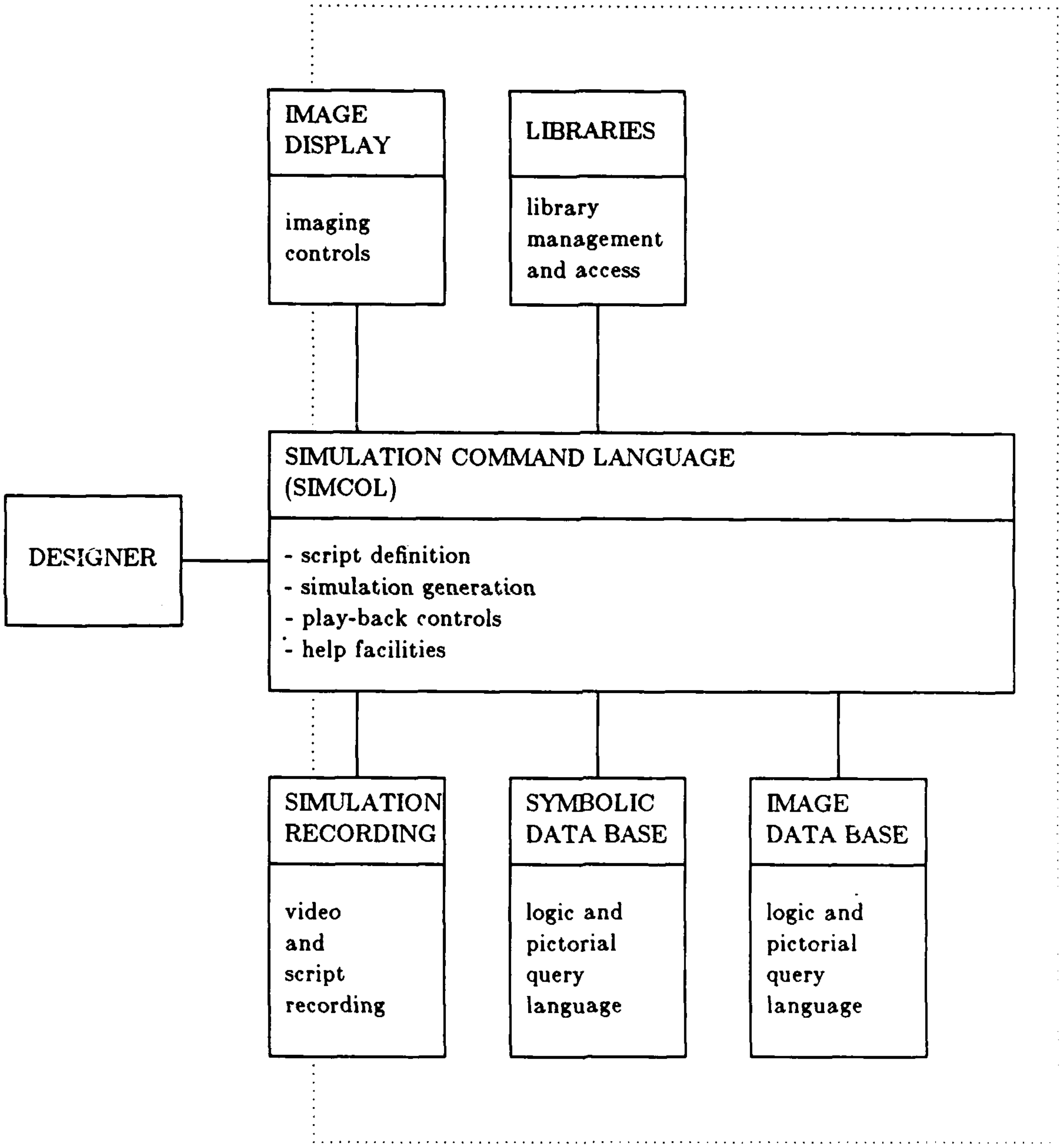


Figure 3.4-1: OVERVIEW OF FUNCTIONAL CAPABILITIES.

invocation of high-level debugging and analysis commands, dynamic definition of simulation monitoring and recording options, simulation flow controls, etc. The monitoring and recording options also play a role in the evaluation of human and technical factors by permitting the playback of experiments recorded on video or computer tape.

HELP FACILITIES

They are available for the designer throughout the interaction with the RQEE and provide on-line documentation of the functional capabilities of the facility.

The developer's role is to enhance the the functional capabilities of RQEE in response to the needs of individual studies without compromising the integrity of its original design. The developer's work falls into two categories. The first is the expansion of the functional capabilities available to the designer through the simulation command language. The second is the enhancement of the database and library holdings, an activity that occurs also during simulation generation.

RQEE contains two databases and several libraries. Their intended contents is summarized in Figure 3.4-3. The libraries contain algorithms used in image processing, image analysis (information extraction), reasoning about abstract geographic information, and visualization of abstract geographic information as synthetic images. Each type of algorithm plays a role in some aspect of the DMA production process. For most part, the intent is to use already proven algorithms and adapt them to the specifics of RQEE. Particular studies may, however, need to develop specific new algorithms for some aspect of the production process.

The image database consists of source images (e.g., Landsat), processed source images which are the result of passing some source image through consecutive steps in the production process along some simulation script (e.g., a source image on top of which the roads recognized so far are overlayed) and synthetic images used to render abstract information in a visual form (e.g., panelling the information extracted from two overlapping photographs). Access to the image database should be provided by means of a logical query language augmented by a pictorial query language.

The symbolic database supplies RQEE with the power and flexibility that will permit even the evaluation of such far-term concepts as the integrated DMA data/knowledge base. Supported by the logical inference mechanisms built in the Prolog language, the symbolic database allows easy representation of simulation scripts, of geographic information abstracted from the source images or supplied by the cartographer, and of various models of world knowledge used in different studies. Rules of reasoning about geographic information, data consistency definitions, and specialized expert systems are seen as part of the world knowledge maintained by the symbolic database and made available (selectively) to the facility user. Here again, both logical and pictorial query languages are needed to provide the access mechanism to the database.

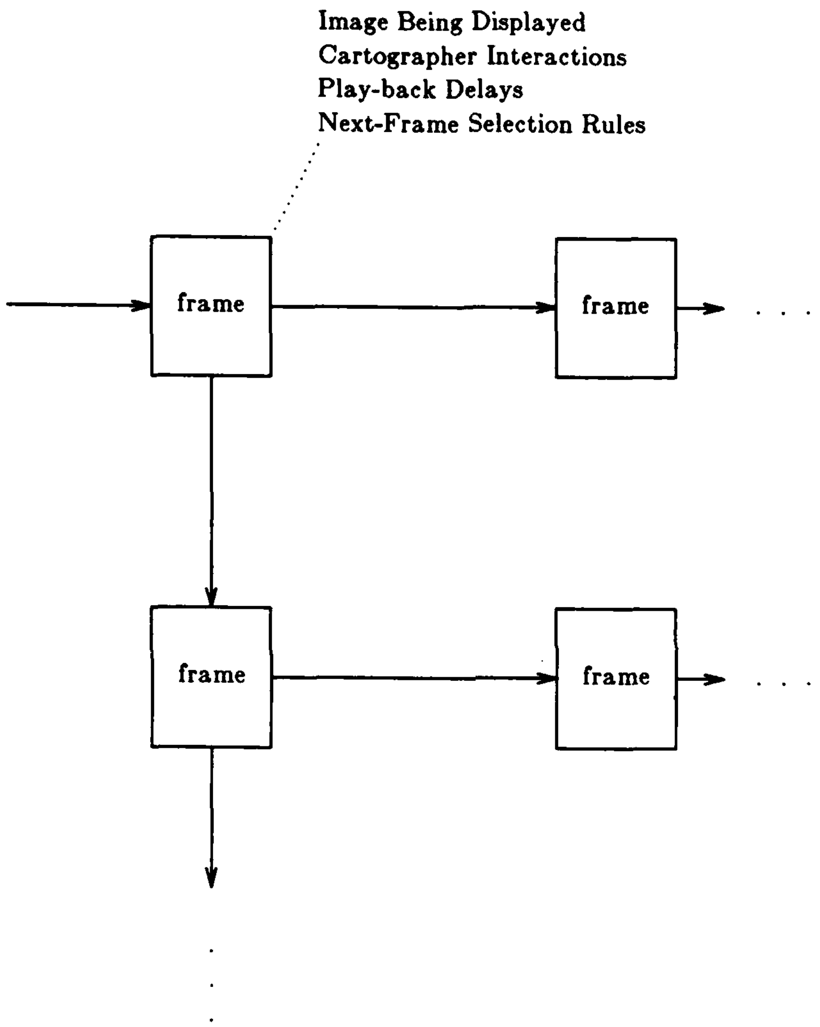


Figure 3.4-2: THE SIMULATION SCRIPT.

---------- SYMBOLIC DATABASE ----------

WORLD KNOWLEDGE

- definitions specific to application domain
- information derivation rules
- semantic constraints
- logical inference rules

BASIC GEOGRAPHIC INFORMATION

- information supplied by the cartographer
- information extracted from images
- qualifiers (spatial, temporal, accuracy, security and source)

SIMULATION SCRIPTS

---------- IMAGE DATABASE ----------

SOURCE IMAGES

OUTPUT IMAGES

- synthetic (derived from logical information)
- processed source images
- digital products

---------- ALGORITHM LIBRARIES ----------

IMAGE PROCESSING ALGORITHMS	-- image to image
IMAGE ANALYSIS (EXTRACTION) ALGORITHMS	-- image to symbolic
GEOGRAPHIC ABSTRACT REASONING ALGORITHMS	-- symbolic to symbolic
GEOGRAPHIC INFORMATION VISUALIZATION ALGORITHMS	-- symbolic to image

Figure 3.4-3: CONTENTS OF THE LIBRARIES AND DATA BASES.

3.4.2. Computing Resources

This section considers a potential hardware/software configuration for the RQEE. The description which follows reflects the current state of available computing technology and is based on a preliminary evaluation carried out at Washington University in Saint Louis. Although the work was not carried out under this contract, the results are included in order to provide a clear picture of what might be involved in building such a requiremets engineering environment.

A DEC VAX-11/750 running Berkeley UNIX[1] 4.2 is recommended as the central processor. Analysis of performance of several 750's on campus which are engaged in image processing and similar compute intensive applications indicate that maximum available memory should be installed. The capacity of the 750 for 8 megabytes of physical memory should be approached if not achieved. The substantial image database required as well as the extensive software libraries and modelling databases suggest the need for substantial disk storage. Currently, technology allows for storage in excess of a gigabyte in reasonable technology (e.g. Eagle drives). At the minimum, the system should be equipped with a Floating Point Accelerator to support both image processing and simulation software. An open question, requiring additional analysis is the need for an array processor, e.g. Floating Point Systems or CSPI.

Since image processing and graphics are central to the production processes of DMA, substantial graphics and image processing hardware is necessary. A Gould/DeAnza IP8500 is the choice for display and image processing. At least two user stations are necessary to allow for analysis of user interactions in the production process. Again, maximum memory in the system would provide for the widest flexibility, speed and utility. In addition to a full complement of memory and image processing options, acquisition of DeAnza's high speed image disk for the IP8500 would allow for high-speed processing and database operations within the image processing hardware. Equipment for producing and inputting images on various media would be required. This includes the ability to digitize and produce film and video materials. Lastly, a number of less expensive displays (e.g. Vectrix or AED) may be useful for pure display and plotting purposes.

The software environment to support the RQEE should be based on UNIX. UNIX itself provides a number of tools including standard programming languages and extensive user utilities. Existing image processing and graphics software may be installed and modified for RQEE use and integrated into the simulation and evaluation environment. PROLOG (or a similar logic based language) should be available for the development of the symbolic database. Appropriate software tools and languages may also be needed based on the needs of specific technology evaluation studies (e.g., performance simulation, VLSI design, expert system construction, etc.).

Figure 3.4-4 represents this currently proposed RQEE computing resource configuration.

[1] UNIX is a Trademark of AT&T Bell Labratories.

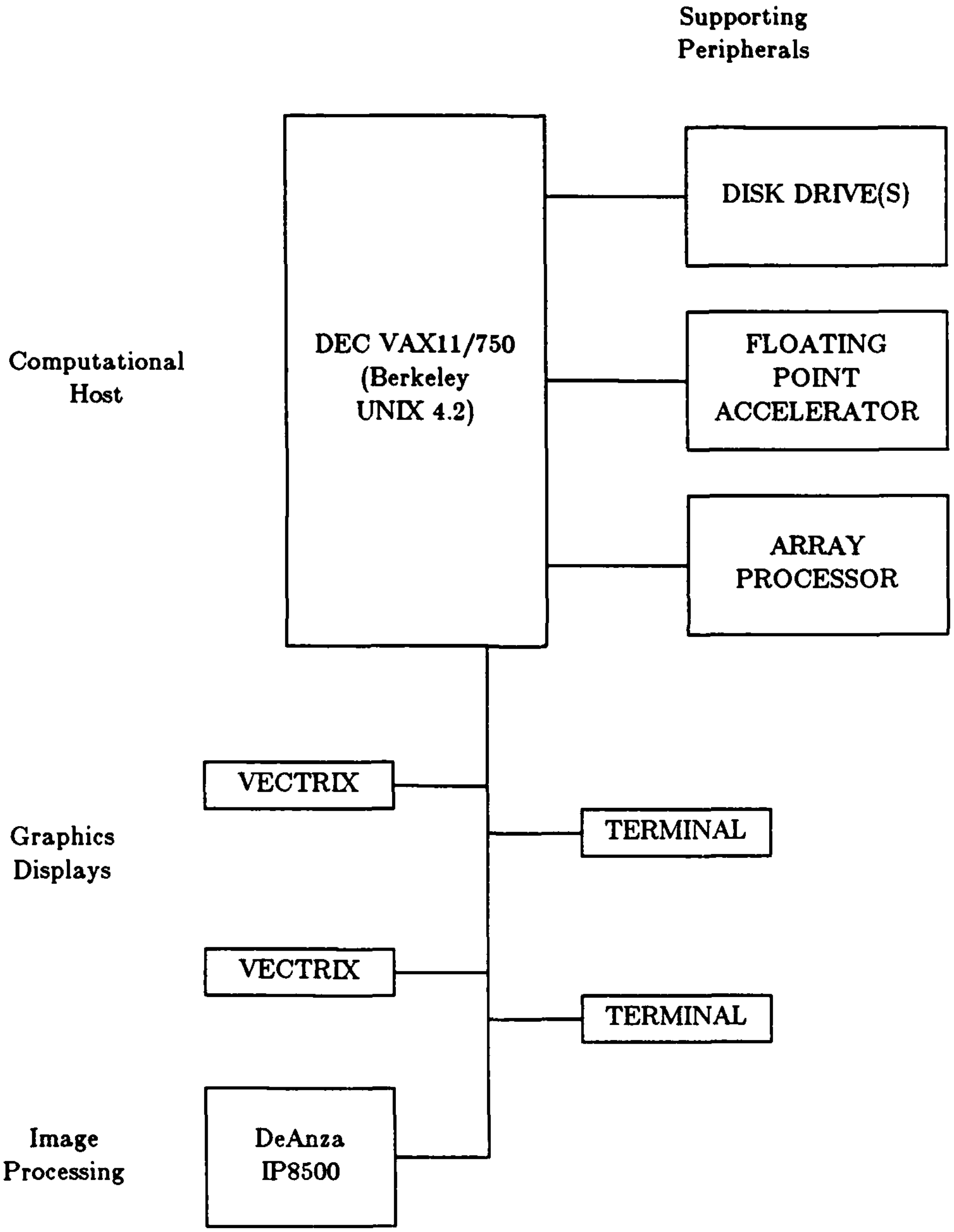


Figure 3.4-4: RECOMMENDED CONFIGURATION.

3.5. FORMAL FOUNDATION

This section establishes a formal foundation for the specification of Geographic Data Processing (GDP) requirements. The emphasis is placed on modelling data and knowledge requirements rather than processing needs. A subset of first order logic is proposed as the principal means for constructing formalizations of the GDP requirements in a manner that is independent of the data representation. Requirements executability is achieved by selecting a subset of logic compatible with the inference mechanisms available in Prolog, a logic-based language which facilitates the expression of abstract relationships. GDP significant concepts such as time, space, accuracy and security classification have been added to the model without losing Prolog implementability or separation of concerns. The rules of reasoning about time, space, accuracy and security may be compactly stated in second order predicate calculus and may be easily modified to meet the particular needs of a specific application. Multiple views of the data and knowledge may coexist in the same formalization.

3.5.1. Introduction

Geographic Data Processing (GDP) systems are computer based systems that store and process information traditionally represented in the form of maps [NAGY79]. Within this broad application area, specialization to particular sets of requirements has led to great variability in the nature of the GDP systems. Some, such as the Landsat data bank [ZOBR81], are primarily repositories of image data while others store no images but use maps to render geographically significant information such as census data (e.g., DIME [SILV77]). The most ambitious undertakings are in the areas of automated cartography and weapon systems support. (The Defense Mapping Agency has the primary responsibility for providing these services for both civilian and military uses.)

The high investment, risk and complexity associated with the development of GDP systems has kept their number relatively small. This, in turn, has resulted in a shortage of development methodologies and tools supporting specificly geographic data processing, especially when compared with what is available today in business data processing. Although it is generally accepted that the development of complex, mission critical, production critical and long-lived systems must start with a validated statement of requirements, the shortage is particularly acute in this area. The lack of any material dealing with GDP requirements in the November 1981 issue of *Computer* which was dedicated to "pictorial information systems" is not accidental.

Here are some of the features we perceive to be important in a GDP requirements specification and validation methodology:

(1) High degree of formality—it assures both precision and executability of the requirements;

(2) Data representation independence—it leads to increases in the stability of the requirements since the data representations may change from one version of the system to the next;

(3) Ability to specify a broad range of functional requirements including image processing, image analysis, product generation, etc.

(4) Compatibility with conceptual models used for human interface definitions and data base specifications;

(5) Support for human factors evaluation;

(6) Support for performance constraints clarification such as response time bounds and productivity estimates;

(7) Ability to specify and evaluate functional enhancements to the system independently of its current realization.

The goal of this section is to establish the formal foundation for a GDP requirements specification and validation methodology meeting many of the objectives stated above. The approach may be summarized as follows. A subset of first order logic is proposed as the principal means for constructing a GDP requirements formalization which is independent of the data representation. Requirements executability is achieved by selecting a subset of logic compatible with the inference mechanisms available in Prolog. Application specific concepts such as time, space, accuracy and security classification have been added to the model without losing Prolog implementability. The methodology uses the formalization and its Prolog realization to build and evaluate models of the system's data, functionality and human interfaces.

The formalism assumes that a GDP system perceives the real world as a collection of abstract objects corresponding to different users' views of various geographic entities. Information about an individual object or a group of objects is captured by facts formally defined as first order predicates. The facts fall into two categories: basic facts which are simply assumed to be true and virtual facts which are defined in terms of other basic and virtual facts. For instance, "Saint Louis is a large city" could be a basic fact, when stated as such, or a virtual fact, if established by using a rule that says "every city with a population in excess of a million people is a large city."

General knowledge about the real world is captured in the definition of application dependent semantic domains, constraints, meta-facts and meta-constraints. Semantic domains usually consist of a set of valid values and permitted operations over them. Temperature is an example of a semantic domain. Individual temperature values may be used to further qualify facts as in "Saint Louis average winter temperature is 45 F." Two semantic domains that are essential to dealing with geographic concepts, space and time, have been built into the formalism. Without ruling out alternate views of space and time, predefined spatial and temporal operators are provided. They allow one to state that a fact is true at some point in time and in a particular place. These operators are used to define other useful concepts such as spatial resolution and seasonal time, to name only two. Accuracy and security classification are two other semantic domains discussed in this section.

Constraints define possible semantic inconsistencies between facts. A bridge, for instance, may not be both open and closed at the same instant of time. The meta-facts and meta-constraints play roles similar to their non-meta counterparts but they are expressed using second order logic. This allows them to deal with knowledge that transcends the specifics of individual facts. The statements "any unstated fact is assumed to be false" and "a fact may not be both true and false at the same instance of time" illustrate the nature of meta-facts and meta-constraints, respectively.

Multiple views of data held by different users and data reinterpretation that occures with the passage of time are captured by the model concept. A model is a grouping of facts and constraints that together define a particular interpretation of the data and of the laws governing the real world.

The remainder of this section discusses: the representation of simple factual data; the representation of world knowledge; and spatial, temporal, accuracy and security qualification of facts.

3.5.2. Representation of Simple Factual Data

This section introduces the concept of object and gives the definitions for basic and virtual facts. Together they provide the means by which geographic entities are identified, attributes are attached to these entities, and relations between them are explicitly stated.

3.5.2.1. Objects and object designators

The notion of object is a primitive concept that allows an individual or a community to distinguish between different geographic entities whose existence they acknowledge. One system user may perceive an island as a single object which is characterized by elevation and vegetation cover while another user may identify two objects: one characterized by elevation and a second one characterized by vegetation cover.

Each object is uniquely identifiable by an object designator, a constant from the finite set

$$\text{OBJ} = \{x_1, x_2, \ldots, x_n\}.$$

Facts refer to the objects they characterize by means of the object designator. It must be pointed out, however, that the object designator is only a formalization convenience since one is expected to reference objects through the properties they exhibit:

"the object X having the property q"

where X is a variable ranging over the set OBJ. (The name of an object, i.e., "St. Louis" is considered to be a property of the object.)

3.5.2.2. Basic facts

A basic fact is a property known to be true about some object or an n-ary relation existing between several objects. This may be formally stated using an n-ary predicate q from the set PR of constant predicates. The set of basic facts is denoted by BF. One instance of BF may assert the following facts

$$\text{road}(x_1),\ \text{road}(x_2),\ \text{road\_intersection}(x_1,x_2)$$

but not

$$\text{road}(x_1),\ \text{road}(x_2),\ {\sim}\text{road\_intersection}(x_1,x_2)$$

where "~" denotes logical negation.

Although logical negation is desirable, the inference mechanism of Prolog (our target implementation language) currently disallows its use. Thus, instead of asserting that a bridge is open or not open (i.e., open(x) or ~open(x)) one has to state that the bridge is open or closed (i.e., open(x) or closed(x)). This creates the possibility for logical contradiction when one asserts that the bridge is both open and closed since the inference mechanism assumes that the two predicates are independent of each other. As shown later, this problem may be alleviated by adding a constraint that states that the bridge may not be both open and closed Nonetheless, the absence of logical negation is a nuisance.

3.5.2.3. Virtual facts

A virtual fact is a fact whose truth value is determined by the truth of other facts, basic or virtual. A virtual fact is asserted by giving its definition. One may state, for instance, that a bridge has a known status if the bridge is known to be either open or closed:

$$(\text{for-all } X)\text{: } (\text{bridge}(X) \wedge (\text{open}(X) \mid \text{closed}(X)) \rightarrow \text{known\_status}(X))$$

For a particular bridge, say x, the fact known_status(x) is true if open(x) or closed(x) is in BF.

We use DVF to denote the set of virtual fact definitions, VF to denote the set of all true virtual facts and TF to denote the set of all true facts, i.e., TF = BF U VF. Any fact that is not in TF is said to be *undefined*, in conformity with what is generally known as the *open world assumption*. By contrast, the *closed world assumption* states that any fact that is not in TF is automatically false. Such an assumption, extensively used in the database area, would be inappropriate for mission critical systems. Moreover, the *closed world assumption* may be explicitly stated, if necessary.

The definition of a virtual fact assumes the general form

$$(\text{for-all } X_1)\ldots(\text{for-all } X_n)\text{: } (F(X_1,\ldots,X_n) \rightarrow q(X_1,\ldots,X_m))$$

where X_i are variables over OBJ, F is a formula defined below, X_1 through X_n are free variables in F, q is a constant predicate and X_1 through X_m are some of the free variables in F. For a more compact notation, however, the general definition could be rewritten as

$(\text{for-all } X_i)\text{: } (F(X_i) \rightarrow q(X_k))$

with i in I, k in K, and K a subset of I.

Using this compact notation, the formula F may be defined recursively as follows:

$F(X_i) ::= q_1(X_i)$
with q_1 in PR;

$::= (F_1(X_{i1})\ \hat{}\ F_2(X_{i2}))$
with i1 in I1, i2 in I2, and I1 U I2 = I;

$::= (F_1(X_{i1})\ \mid\ F_2(X_{i2}))$
with i1 in I1, i2 in I2, and I1 U I2 = I;

$::= (F_1(X_i)\ \hat{}\ (\text{for-all } X_j)\text{: } (F_2(X_{i2},X_j) \rightarrow F3(X_{i3},X_j)))$
with i2 in I2, i3 in I3, I2 U I3 subset of I,
j in J, and j not in I;

$::= (F_1(X_i)\ \hat{}\ \text{not}(F_2(X_{i2})))$
with i2 in I2, and I2 subset of I;

where i in I, the symbols F_1, F_2 and F3 are instances of F, and the "not" operator is not the logical negation but a test that a formula may not be shown to be true. Here again, the restrictions imposed over the definition of F are motivated by the need to achieve implementability of the formalism.

The examples below provide simple illustrations of virtual fact definitions:

(a) A road is open if all bridges on that road are open.

(for-all X): (road(X) ^
(for-all Y): (bridge(Y,X) → open(Y)) → open_road(X))

Note: this definition is equivalent to stating that
"there is no bridge which is not known to be open"
but since existential quantifiers are not permitted
the definition required some reformulation.

(b) A bridge that is not known to be open is assumed to be closed.

(for-all X): (bridge(X) ^ not(open(X)) → closed(X))

Note: the alternate form (for-all X): (not(open(X)) → closed(X))
is illegal because the range of X must be
limited to some well-defined finite set.

(c) A bridge that is open or closed has a known status.

(for-all X): (bridge(X) ^ (open(X) ¦ closed(X)) → known_status(X))

Note: this definition is actually equivalent to
(for-all X): (bridge(X) ^ open(X) → known_status(X))
(for-all X): (bridge(X) ^ closed(X) → known_status(X))

This concludes the discussion of virtual facts. More complex formulations of world knowledge are covered in the next section.

3.5.3. World Knowledge Representation

Semantic domain, constraint, model, meta-fact, meta-constraint and meta-model are the key concepts introduced in this section. They are the means by which general knowledge about the world, rather than specific facts, may be expressed. Even systems that may not be called "knowledge-based" encapsulate in their functional requirements a certain user view (i.e., knowledge) of the world. It is our conjecture that functional enhancements are the result of changes in this world knowledge and the difficulty of specifying system enhancements is a consequence of having the world knowledge implicit in the system functionality. Explicit specification of the world knowledge, as part of the system requirements, is expected to reduce the occurrence of inconsistencies in the requirements specifications, to unify and simplify the specification of the individual functions supported by the system, to improve enhanceability and to lead to a uniform treatment of GDP requirements whether or not the systems are "knowledge-based".

3.5.3.1. Semantic domains

A semantic domain is defined as a set of values and operations over them, i.e., an abstract data type. These values are used to qualify properties ot objects but they themselves may not be treated as objects. Semantic domain values do not stand for geographic entities and are not necessarily finite in number.

The value 50 of the semantic domain temperature might be used, for instance, in a fact such as

"the average temperature in Saint Louis is 50 F."

To state this formally, one needs to extend the scope of the predicates in PR to include both object designators and semantic domain values:

average_temperature(50)(STL)

For clarity sake, the notation distinguishes between the two types of arguments.

Each semantic domain D (member of the set SD) is specified by its characteristic function D(d) and the set of permissible operations. The characteristic function returns true whenever the argument is in the respective semantic domain. The set of permissible operations is domain specific. While temperatures may be added, subtracted and compared for relative position in some total order, crop types may only be tested for identity. Some operations may relate values from different semantic domains thus providing, among other things, type conversion facilities. One such example is the correspondence between Fahrenheit and Celsius scales.

Semantic domain specific operations that return boolean values (e.g., the characteristic function) may be used in the definition of virtual facts and constraints as if they were facts with the proviso that the value "false" is interpreted as "not provable." Furthermore, for infinite domains, any variable supplied as argument to the operation should be preceded by an earlier occurrence of the same variable in a fact, when possible. The motivation rests again with the limitations of Prolog, which evaluates expressions from left to right. For instance, if one tries to compute all the recorded temperatures in accordance with the definition

$$\text{(for-all X,Y): (TEMPERATURE(X)} \wedge \text{average\_temperature(X)(Y)} \rightarrow \text{recorded(X))}$$

when the set of temperatures is not finite, the evaluation may fail or take forever. The same definition, however, rewritten as

$$\text{(for-all X,Y): (average\_temperature(X)(Y)} \wedge \text{TEMPERATURE(X)} \rightarrow \text{recorded(X))}$$

avoids this problem.

The role of semantic domains in the definition of semantic consistency is discussed next under constraints while ways of "reasoning" about facts involving certain semantic domains receive extensive coverage under the headings of spatial, temporal, accuracy and security qualification of facts.

3.5.3.2. Constraints and semantic consistency

Facts and constraints correspond, to some extent, to two concepts widely used in the database literature: extension and intension (e.g., [MINK78]). A fact states a property attributed to some object while a constraint states what properties the object may or may not have, one represents observations while the other captures general laws.

Constraint definitions (the set SC) assume a form very similar to virtual fact definitions

$$\text{(for-all } X_i\text{): (F(}X_i\text{)} \rightarrow \text{ERROR(type\_of\_violation,}X_k\text{))}$$

with i in I, k in K, and K subset of I. If the distinguished predicate ERROR, not included in PR, takes the value true for some set of arguments, the facts in TF are said to be *inconsistent* with respect to the set of constraints SC.

Constraints may be employed in enforcing a many-sorted logic and general laws of nature and society. In a many-sorted logic, the arguments of each predicate may be restricted to certain predefined domains. (See [MINK78] for a brief overview.) An anomalous fact such as average_temperature(green)(STL),

for instance, may be flagged by defining the constraint

(for-all X,Y): (average_temperature(X)(Y) ^ not(TEMPERATURE(X))
→ ERROR(improper_temperature,X))

An example of a general law is "each state has only one capital city" which may be written as

(for-all X,Y,Z): (capital_of(X,Z) ^ capital_of(Y,Z) ^ not(X=Y)
→ ERROR(two_capital_cities,Z))

Because agreement on general laws is only rarely achievable in the community at large, constraints could be rendered useless unless one acknowledges that they are valid only relative to someone's point of view. This is accomplished by the introduction of the model concept below.

3.5.3.3. Models and knowledge management

The motivation for adding the concept of model to the formalism rests with the recognition that users have different views of the same data, that data is often reinterpreted because of changes in the application area, that simplifying assumptions about the data are frequently made for the purpose of satisfying particular data processing requirements and, above all, that general knowledge about the world is in a continuous state of flux. To deal with these hard realities of geographic data processing, facts are said to be true with respect to some point of view which is identified by using a model name as a qualifier for the fact:

celsius'freezing_point(0)(x).

This basic fact, for instance, states that the freezing point associated with the object x is 0 degrees, if one assumes the Celsius scale.

Since, in general, models are not independent of each other, being able to relate true facts in some model to true facts in the others is very important. For this reason, a model may be used to qualify any desired fact or constraint and their definitions may involve facts qualified by any other model. This is illustrated, for instance, by the relation between the freezing point in the Celsius and Fahrenheit scales:

(for-all X): (celsius'freezing_point(0)(X)
→ fahrenheit'freezing_point(32)(X))

Furthermore, model names may be universally quantified as in

(for-all X,Y,Z): (X'freezing_point(Y)(Z) ^ not(TEMPERATURE(Y))
→ semantics'ERROR(improper_temperature,Y))

By definition, any fact or constraint violation that is not explicitly qualified by some model is associated with a default model w, i.e.,

$q(x) \rightarrow w'q(x)$

The way to limit the universe of discourse to specific models is considered next.

3.5.3.4. World view

If MD is used to denote the (finite) set of all models, the world view, WV, is defined as any non-empty subset of MD. By specifying a particular world view, one states that any fact that is true only with respect to models not present in WV is not of interest and, therefore, is assumed to be not provable. Because the same rule is applied to constraints, a constraint violation may occur in one world view but not in the other. A world view that exhibits no constraint violations is called consistent.

A GDP system may present different world views to different users. When no world view is specified WV = {w} is assumed. The characteristic function for the world view is WV(X). The facts which are true in the current world view may be referred without specifying any particular model. In other words, when the world view is WV = $\{w,m_1,m_2\}$, for instance,

$q_0(x) \rightarrow q_1(x)$

is the same as

$(w'q_0(x) \mid m_1'q_0(x) \mid m_2'q_0(x)) \rightarrow w'q_1(x)$.

This concludes the discussion of features based on the first order predicate calculus. The formalization of more abstract knowledge, captured by meta-facts and meta-constraints, takes us to second order predicate calculus.

3.5.3.5. Meta-facts

Meta-facts provide the means by which one specifies new axioms and rules of inference, usually related to reasoning about a specific semantic domain. They involve higher order knowledge which, as a rule, tends to be independent of particular predicates. Predicate independence is accomplished by employing second order logic. In our formalization, this leads to the use of universally quantified variables that range over the set of defined predicates. In all other respects, the syntax of meta-facts is identical to that of facts

To enable the writing of the meta-rules, three characteristic functions are added to the formalization: PR for the set of defined predicates (for exclusive use in meta-facts and meta-constraints), OBJ for the set of objects, and MD for the set of models. Furthermore, these characteristic functions will check not only individual entities for membership in the respective set but lists of entities. This is because, for generality sake, meta-facts often refer to arbitrarily long lists of arguments (e.g., object designators) by single variables. One example is provided by the statement that "a fact that is true in w is taken to be true in every model:"

(for-all M,Q,X): $(w'Q(X) \rightarrow M'Q(X))$

Another illustration is the statement of the closed world assumption, i.e., "any fact not known to be true is assumed to be false:"

$$(\text{for-all } M,Q,X)\text{: } (M'Q(X) \rightarrow M'Q(\text{true})(X))$$

$$\begin{aligned}(\text{for-all } M,Q,X)\text{: } (MD(M) \wedge PR(Q) \wedge OBJ(X) \\ \wedge \text{ not}(M'Q(\text{true})(X)) \rightarrow M'Q(\text{false})(X))\end{aligned}$$

The use of meta-facts in defining rules of inference for reasoning about space, time, security classification and data accuracy is illustrated in later sections. Dealing with last two semantic domains will require the ability to examine not only facts but the way they are derived. Consequently, the domain of valid (first order) formulas must be made available. Under the simplifying assumption that each virtual fact q has a unique definition, its definition may be denoted by DEF(q). For instance, a rule that requires "any object that appears in a predicate whose definition is sensitive to be classified" may be written as follows:

$$(\text{for-all } Q,X)\text{: } (Q(X) \wedge SENSITIVE(DEF(Q)) \rightarrow CLASSIFIED(X))$$

3.5.3.6. Meta-constraints

Meta-constraints play the same role as constraints but at the meta-level. Syntactically they follow the same format but are permitted to take advantage of the power available for meta-facts. This is illustrated by the formalization of the statement "no fact may be both true and false:"

$$\begin{aligned}(\text{for-all } M,Q,X)\text{: } (M'Q(\text{true})(X) \wedge \\ M'Q(\text{false})(X) \rightarrow M'ERROR(\text{logical\_contradiction},Q,X))\end{aligned}$$

This meta-constraint and the related meta-facts introduced earlier (together referred as meta-rules) may be treated as a group and used only when needed. The way this is done is explained next.

3.5.3.7. Meta-models

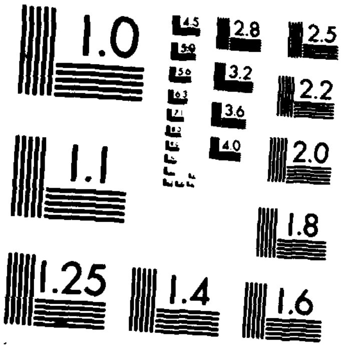
One or more semantic domains, their associated operations and pertinent meta-rules may be grouped together to form a meta-model. The separation into meta-models enables the experimentation with different rules of inference without having to change the formalization. A potential source of formalization errors is thus eliminated while also increasing the productivity of the requirements specification and validation process.

3.5.3.8. Meta-view

The meta-view consists of all the meta-models in use at one particular point in time. The ability to specify a subset of all the defined meta-models is needed to address specific concerns of the requirements evaluation process and to compare alternate formalizations of the semantic domains. Separation of concerns and incremental

AD-A173 599 LONG-RANGE TECHNOLOGICAL IMPACT ON COMPUTER-AIDED PRODUCT DEVELOPMENT AT (U) WASHINGTON UNIV ST LOUIS MO DEPT OF COMPUTER SCIENCE G ROMAN ET AL JUL 86 2/3

UNCLASSIFIED RADC-TR-86-49 F30602-83-K-0065 F/G 9/2 NL


1.0
4.5
5.0
5.6
6.3
7.1
8.0
9.0
2.8
2.5
3.2
2.2
3.6
4.0
1.1
2.0
1.8
1.25
1.4
1.6

building of the requirements formalization are also encouraged.

The remaining sections develop sample meta-models for several important semantic domains.

3.5.4. Spatial Qualification of Facts

The concept of space is quintessential in geographic data processing. Geographic entities exist in space and each location on the earth surface corresponds to a position in the 3D space occupied by the planet. This section provides one particular formalization of the space concept. It starts by introducing the notion of absolute space as an abstraction of the physical space. Absolute space is an infinite semantic domain defined in terms of real numbers, traditional operations over reals and three domain specific operations: the characteristic function, distance and direction.

Absolute space is later used to define the concept of logical space. The latter captures the idea of finite resolution which is more representative for the way geographic data processsing is actually carried out. Several spatial operators are also defined using a framework similar to the one already established by positional logic. They allow one to state that an individual fact is realized only at particular points in the logical space. The relation between the spatial operators and their use in the definition of certain space dependent concepts is detailed under the last two headings of this section.

3.5.4.1. Absolute space

The absolute space is an abstraction of the coordinate system being used. The presentation, however, is not dependent on any particular coordinate system since changes in the coordinate system definition are expected to affect only the definition of the absolute space and not the rules of reasoning about spatial properties.

Given the set of real numbers, RN, and the traditional operations over reals, the definition of the absolute space takes the following form:

ASPACE = (XCRD, YCRD, ZCRD, dist, dir)

where XCRD, YCRD and ZCRD are subsets of RN,

$p \in (XCRD \times YCRD \times ZCRD) \rightarrow ASPACE(p)$
when ASPACE is used to denote the characteristic function

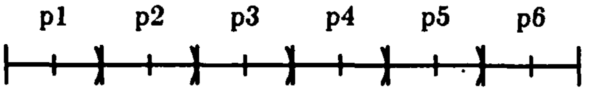
$dist : RN^6 \rightarrow RN$ (the distance function)
$dist(p,p) = 0$
$dist(p_1,p_2) = dist(p_2,p_1) \geq 0$
$dist(p_1,p_3) \leq dist(p_1,p_2) + dist(p_2,p_3)$

dir : $RN^6 \rightarrow RN^2$ (the direction function)
dir(p,p) = undefined
(dist(p_0,p), dir(p_0,p)) is unique for each p, given a fixed p_0.

Polar, Cartesian, Universal Transverse Mercator and other coordinate systems could be easily shown to satisfy these properties.

3.5.4.2. Logical space and finite resolution

The logical space is defined as a discrete subset of an absolute space. A convenient way to specify a logical space is to define a mapping that takes each point from the absolute space into a point of the logical space. This function is called the *resolution function*. A proper resolution function partitions the absolute space into intervals in a manner analoguous to the partitioning of the real line segment below:



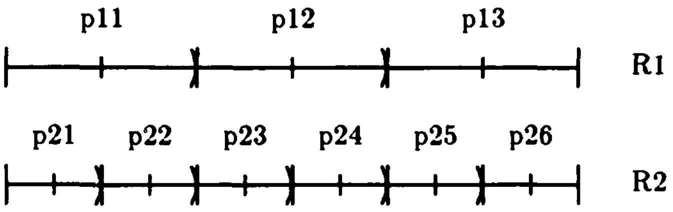
where each p_i is a point in the logical space. The point p_i is also called the representative point for the corresponding interval because all the points in the interval are mapped into p_i. For the 3D case, the domain of each coordinate must be partitioned in this manner. Although the intervals are usually uniform in size this is not a necessary condition for defining a proper resolution function.

In the remainder of this section, the symbol R is used to denote both the resolution function for an individual logical space and the respective logical space with the intended interpretation being determined by the context. Moreover, a single common absolute space is assumed for all the logical spaces required by the exposition.

Under these assumptions, a logical space R_2 is said to be a refinement of another logical space R_1, (i.e., $R_2 >> R_1$), if and only if

$$(\text{for-all } p_1,p_2)\text{: } (R_2(p_1)=R_2(p_2) \rightarrow R_1(p_1)=R_1(p_2))$$

The figure below is a case in point:



This definition is useful in relating properties stated with respect to different logical spaces.

3.5.4.3. Spatial operators

The spatial operator provides the notational means to specify that a particular property is true at some position in space. The simplest spatial operator originates in work on positional logic [RESC71]. It merely states that "property q(y) of object list x is true at position p:"

@p q(y)(x).

(For the sake of clarity, the existence of the argument list over semantic domains, i.e., y, is omitted wherever its presence is superfluous.) The operator '@' is allowed to qualify only facts and not entire formulas; the position may be a universally quantified variable; the space is treated as any other semantic domain; and the resolution function may be applied to the position as in @R(p) q(x).

The semantics of the *simple spatial operator* are captured by the following meta-facts:

(for-all P,Q,X): (@P Q(X) → Q(P)(X))
position may be treated as an additional qualifier

(for-all P,Q,X): (Q(X) → @P Q(X))
space independent facts are true at every point in space

(for-all P_x,P_y,P_z,Q,X): (@(P_x,P_y) Q(X) → @(P_x,P_y,P_z) Q(X))
facts specified in an elevation independent manner are
taken to be true at any elevation one might consider

To illustrate the use of the spatial operator we provide the definitions for two facts, one basic the other virtual:

@p vegetation(pine)(hill)

```
(for-all $P_0$,X):
   ( @$P_0$ elevation(X) ^
     (for-all $P_1$):
        ( @$P_1$ elevation(X) ^
          (dist($P_0$,$P_1$) < $\delta$) → ($P_0$.z > $P_1$.z))
              → @$P_0$ elevation_peak(X))
```

where P.z refers to the z coordinate of P=(P_x,P_y,P_z).

Although the resolution function may be used to assure that the specified position is within some required logical space, the simple spatial operator is independent of the concept of logical space. This is not so with its extensions proposed in this-section: the area uniform, area sampled and area averaged operators. They are attempts to formalize the nature of the information being maintained when working with finite resolution

The *area uniform operator* states that a property that is true for some point of a logical space is true for each point of the absolute space mapped into that particular point of the logical space. Formally, this is stated as:

$(\text{for-all } R,P,P_0,Q,X)\text{: } (@_u[R]P\ Q(X)\ \hat{}\ R(P)=P_0 \rightarrow Q(R,u,P_0)(X))$

the operator introduces additional qualifiactions

$(\text{for-all } R,P_0,P,Q,X)\text{: } (@_u[R]P_0\ Q(X)\ \hat{}\ R(P)=R(P_0) \rightarrow @P\ Q(X))$

the property is true for all points in the area

$(\text{for-all } R_1,R_2,P_1,P_2,Q,X)\text{:}$
$(\ (R_2 >> R_1)\ \hat{}$
$@_u[R_1]P_1\ Q(X)\ \hat{}\ R_1(P_2)=R_1(P_1) \rightarrow @_u[R_2]P_2\ Q(X))$

the property is inherited by the higher resolution subareas of a low resolution area

$(\text{for-all } R_1,R_2,P_1,Q,X)\text{:}$
$(\ (R_2 >> R_1)\ \hat{}$
$(\text{for-all } P_2)\text{:}$
$(R_1(P_2)=R_1(P_1) \rightarrow @_u[R_2]P_2\ Q(X)) \rightarrow @_u[R_1]P_1\ Q(X))$

the property is acquired by a low resolution area if all its high resolution subareas share the same property

(Note: (1) Ri is treated here as a variable that ranges over the set of resolution functions. This is the case any time R is quantified. (2) In the second meta-fact above, because there is an infinity of points P satisfying the condition $R(P)=R(P_0)$, any attempt to find them is bound to fail unless the meta-fact is used in a context where the set of values taken by P is finite.)

To illustrate the use of the area uniform operator, let us consider a low resolution photograph including a lake. If an image point is discerned to be part of the lake, it follows that all the individual points covered by it are also part of the lake. Therefore, one would be justified in writing

$(\text{for-all } P)\text{: } (@P\ \text{color(blue)(image)} \rightarrow @_u[R]P\ \text{lake}(x))$

if the points on or outside the lake boundary never appear pure blue.

The area uniform operator also provides the means by which the underlying absolute space may be replaced by a finite resolution space for applications where this substitution is appropriate, e.g., when a maximum target resolution may be determined. A meta-fact of the form

$(\text{for-all } P,Q,Y,X)\text{: } (@P\ Q(Y)(X) \rightarrow @_u[R]P\ Q(Y)(X))$

is all that is required to accomplish the transition to a finite resolution view of the world. For semantic domains where each point may be assigned a unique value, the uniqueness is guaranteed by the meta-model where the domain is defined and no additional meta-constraint is needed here.

The *area sampled operator* may be employed whenever one needs to state that there is at least one point in the area having a certain property but the actual point is not necessarily known. The meta-facts defining this operator are the following:

$(\text{for-all } R,P,P_0,Q,X)\text{: } (@_s[R]P\ Q(X)\ \hat{}\ R(P)=P_0 \rightarrow Q(R,s,P_0)(X))$

the operator introduces additional qualifications

$(\text{for-all } R,P,Q,X)\text{: } (@P\ Q(X) \rightarrow @_s[R]P\ Q(X))$

the area acquires the sample if any point in the area has the property

$(\text{for-all } R_1,R_2,P_1,P_2,Q,X)\text{:}$
$(\ (R_2 >> R_1)\ \hat{}$
$@_s[R_2]P_2\ Q(X)\ \hat{}\ R_1(P_2)=R_1(P_1) \rightarrow @_s[R_1]P_1\ Q(X))$

the area acquires the sample if any subarea has it

Continuing the example, the points corresponding to the lake border may be characterized as covering some part of the lake:

$(\text{for-all } P)\text{: } (@P\ \text{color(part\_blue)(image)} \rightarrow @_s[R]P\ \text{lake}(x))$

The *area average operator* allows the association of a value to a point in the logical space with the understanding that the respective value characterizes not the point but the area it represents. When working with finite resolutions, one deals with average values rather than precise measurements. The semantics of the area average operator are captured by the following meta-facts:

$(\text{for-all } R,P,P_0,Q,Y,X)\text{: } (@_a[R]P\ Q(Y)(X)\ \hat{}\ R(P)=P_0 \rightarrow Q(R,a,P_0,Y)(X))$

the operator introduces additional qualifiactions

$(\text{for-all } R_1,R_2,P_1,Q,Y_0,X)\text{:}$
$(\ (R_2 >> R_1)\ \hat{}\ R_1(P_1)\ \hat{}$
$(Y_0 = \text{average}(\text{"}Y\text{"},\text{"}R_1(P_2) = P_1\ \hat{}\ @_u[R_2]P_2\ Q(Y)(X)\text{"}))$
$\rightarrow @_a[R_1]P_1\ Q(Y_0)(X))$

the average may be computed if values are known for each subarea

$(\text{for-all } R_1,R_2,P_1,Q,Y_0,X)\text{:}$
$(\ (R_2 >> R_1)\ \hat{}\ R_1(P_1)\ \hat{}$
$(Y_0 = \text{average}(\text{"}Y\text{"},\text{"}R_1(P_2) = P_1\ \hat{}\ @_a[R_2]P_2\ Q(Y)(X)\text{"}))$
$\rightarrow @_a[R_1]P_1\ Q(Y_0)(X))$

the average may be computed if average values are known for each subarea

where the function called "average" determines which points in the logical space R_2 are mapped into the point P_1, by the resolution function R_1, and for these points computes the average value of any of the arguments of the fact Q for a particular object or group of objects. The average is weighted if the resolution functions partition the space in unequal areas. (We assume this not to be the case.)

The concept of average makes sense only when the values of the semantic domain involved in the averaging computation are of numeric nature and each point may have associated with it a single value. The lake temperature in some area may be

$@_u$[R]p temperature(34)(lake)

or

$@_u$[R]p temperature(39)(lake)

but not both, while it is conceivable that the lake may be uniformly covered, in some area, by two different types of aquatic vegetation, e.g.,

$@_u$[R]p vegetation(a)(lake)

and

$@_u$[R]p vegetation(b)(lake)

In the former case one may average temperatures while in the latter the vegetation types may not be averaged. If values are not numeric and unique for each point, the averaging function fails. Furthermore, the existence of a unique average is enforced by the following meta-constraint being added to the definitions introduced so far:

(for-all R,P,Q,Y_1,Y_2,X):
(${@_a}$[R]P Q(Y_1)(X) ^ ${@_a}$[R]P Q(Y_2)(X) ^
not(Y_1=Y_2) → ERROR(distinct_averages,Q,P)

The four spatial operators introduced in this section provide the ability to specify that a fact is true at a specific point, at some point of a particular area, over all points of a particular area and on the average over some area. They also establish a framework which is instrumental in defining a variety of concepts that reflect spatial properties of geographic entities. This use of the spatial operators is illustrated next.

3.5.4.4. Formalization of spatial properties

This subsection provides examples of how spatial operators may be used to define formally geometric properties of individual objects, spatial relations between objects and the loss of information that takes place when moving to increasingly lower levels of resolution. The ability to formally define these properties is an indicator of the spatial operators' power and usefulness and leads to the development of very powerful primitives that ought to simplify the formalization task.

Geometric properties, as defined here, include type of geographic feature (point, line, area, volume), general topology (e.g., single or multiple points), dimensionality (2D or 3D), geometric constraints (e.g., multiple points on a line, on an area, or in some volume), shape (e.g., square) and size (with respect to some metric). The simplest illustration is given by the definition of a point type feature:

$$\begin{array}{l} \text{(for-all } X,Q_1,P_1\text{): } (@P_1\ Q_1(X)\ \hat{}\ \text{not}(Q_1(X))\ \hat{} \\ \quad ((\text{for-all } P_2,Q2\text{): } (@P_2\ Q2(X)\ \hat{}\ \text{not}(Q2(X)) \rightarrow P_1=P_2)) \\ \qquad\qquad\qquad \rightarrow \text{point\_type}(X)) \end{array}$$

Informally, this definition states that all position dependent properties of the object are true at a single point in space.

Spatial relations between objects cover concepts such as relative position, relative orientation, relative size, adjacency (usually, at some given resolution) and overlap. The definition of overlap, for instance, takes the form:

$$\begin{array}{l} \text{(for-all } X_1,X,X_2,Y_1,Y,Y_2,P,Q_1,Q2\text{):} \\ \quad (@P\ Q_1(X_1,X,X_2)\ \hat{}\ @P\ Q2(Y_1,Y,Y_2)\ \hat{} \\ \quad\ \text{not}(Q_1(X_1,X,X_2))\ \hat{}\ \text{not}(Q2(Y_1,Y,Y_2)) \rightarrow \text{overlap}(X,Y)) \end{array}$$

(Note: X_1, X_2, Y_1 and Y_2 represent arbitrary lists of object designators). Since facts formulated in a space independent manner are true at every point in space, they are excluded from consideration. Otherwise, the concept of overlap would become meaningless.

It is generally accepted that a high resolution map provides more information than a low resolution map covering the same area. When the map generation is automated there is the need to specify the nature of the information loss incurred in the process of interpreting the data with regard to a lower resolution than originally formulated. The rules governing this process are a special case of abstraction rules. Four types of abstraction rules have been identified as particularly useful in this context: copying, thresholding, averaging and composition. They refer to the operations applied to properties of the original object in order to determine the properties of the same object when considered at a lower resolution.

Copying rules state the conditions under which a high resolution point passes on its properties to the low resolution point into which it is mapped. The type of property and the size of the object are two possible considerations. Thresholding rules state the conditions under which properties of the high resolution points are ignored during the transition to the lower resolution. A combined copying/thresholding rule may be applied, for instance, to determine the presence of some island on the map:

$$\begin{array}{l} \text{(for-all } R_1,R_2,P,X\text{):} \\ \quad ((R_2 >> R_1)\ \hat{} \\ \quad\ @R_2(P)\ \text{island}(X)\ \hat{}\ (\text{size}(X,R_2) > \delta) \rightarrow @R_1(P)\ \text{island}(X)) \end{array}$$

where R_1 is the logical space having the lower resolution and "size" is a function that determines the number of points covered by some object at a specified resolution.

Averaging rules state the conditions under which the average of the semantic domain values associated with some property of high resolution points that map into the same low resolution point is assigned to the latter. The definition of the average operator, $@_a[R]$, shows one example of how this rule works.

Finally, composition rules are used to generate new properties for the low resolution point based on the properties of the high resolution points that map into it. One example is the determination of the shore line through the use of the following rule:

$$\begin{array}{l}\text{(for-all } R_1,R_2,P_1,P_2,X):\\ \quad (R_1(P_1)=R_1(P_2)\ \hat{}\\ \quad\ @R_2(P_1)\ \text{lake}(X)\ \hat{}\ @R_2(P_2)\ \text{shore}(X) \rightarrow @R_1(P_1)\ \text{shore\_line}(X))\end{array}$$

where R_1 and R_2 are defined as before.

This concludes the discussion of spatial qualification of facts. The next section, covering temporal qualification of facts, shows how some of the concepts and operators introduced so far are also applicable to reasoning about time, with some minor reformulation.

3.5.5. Temporal Qualification of Facts

The contents of this section parallels in intent recent data modelling efforts concerned with the formalization of temporal concepts used in an ad-hoc manner by database designers. However, while Clifford and Warren [CLIF83], for instance, attempt to extend the relational model to include historical relations, we start with a logic-based formalization and thus we are able to assimilate directly some of the work on temporal logic [RESC71], mostly through the adaptation of the spatial operators introduced previously. This approach is possible because, as noted by others [RESC71], temporal logic may be seen as a special case of the positional logic.

In addition to treating time as a uni-dimensional space, this section also considers the issue of reasoning about arbitrary time intervals and cyclic phenomena and proposes an appropriate set of temporal operators.

3.5.5.1. Time as a uni-dimensional space

If time is treated as a uni-dimensional space, the absolute time, ATIME, is reduced to the real line with distance and direction defined as

$$\text{dist}(t_1,t_2) = \text{abs}(t_2-t_1)$$

and

$$\text{dir}(t_1,t_2) = \text{sign}(t_2-t_1).$$

Logical time, in turn, is introduced with the help of the same resolution function R, while potential confusion between logical time and space is avoided by context. The simple temporal operator

$$\&t\ q(x)$$

is defined as the analog of the simple spatial operator. Furthermore, the interval uniform, average, and sampled operators

$\&_u[R]t\ q(x)$
$\&_a[R]t\ q(x)$
$\&_s[R]t\ q(x)$

mimic their spatial counterparts and play identical roles.

The concept of temporal resolution, however, is not as important as its spatial counterpart. More often than not, reasoning about time involves dealing with arbitrary time intervals during which one property or another holds true for some object. The interval uniform operator, as defined so far, while able to capture this notion, is not convenient to use—a separate resolution function would have to be defined for each interval. One way to overcome this impediment is explored next.

3.5.5.2. Time intervals

This subsection extends the scope of the interval uniform operator, introduces a definition for the concept of "now," and discusses two models useful in reasoning about time.

The proposed extension allows one to suply an interval definition in place of the resolution function as in the following examples:

$\&_u[t_1,t_2]\ q(x)$
$\&_u(t_1,t_2]\ q(x)$
$\&_u[t_1,t_2)\ q(x)$
$\&_u(t_1,t_2)\ q(x)$

with t_2 always greater than or equal to t_1.

Because of the similarity between their definitions, only the definition for the closed interval case is provided below.

$$\text{(for-all } T,T_1,T_2,Q,X\text{)}: (\&_u[T_1,T_2]\ Q(X)\ \hat{}\ (T_1 \leq T \leq T_2) \rightarrow \&T\ Q(X))$$

It should also be noted that, when considering finite time resolution, e.g., years, only closed intervals are needed. For instance, the basic fact

$\&_u[1971,1973]$ poor_harvest(x)

states that the harvest was poor from 1971 through 1973, inclusively.

One further complication brought about by dealing with intervals is the need to consider the concept of present moment or "now." Staticly, now is a unique point in time separating past from future. In this context, it is important to be able to tell if some arbitrary point in time is part of the past, present or future. Three functions bearing these respective names could be provided for this purpose:

past(1971) — is provable, the year is 1986;
present(1971) — is not provable;
future(1971) — is not provable;

They are not adequate, however, for dealing with the dynamics of now, i.e., the present becoming past while the future is becoming present.

A special place holder, call it "now," must to be introduced in order to express facts whose truth changes as the present moves into the future. In the simplest case, one must be able to state that some fact is always true in the present as in

$$\&\text{now } q(x)$$

whose semantics is given by

$$(\text{for-all } T,Q,X)\text{: } (\&\text{now } Q(X) \hat{} \text{ present}(T) \rightarrow \&T\ Q(X)).$$

Similarly, one may allow "now" to appear as interval boundaries. The use of unevaluated expressions becomes necessary, however, if one wants to go so far as to permit intervals such as [now=5,now+5].

Further illustrations of the interval uniform operator are provided by the formulation of two models important in reasoning about time. The first one, called in [CLIF83] the *comprehension principle,* is a variation on the closed world assumption. It says that although some fact may not be uniformly true over some interval of interest, it is often expedient to assume that it is:

$$(\text{for-all } T,Q,X)\text{: } (\&T\ Q(X) \hat{} (t_1 \leq T \leq t_2) \rightarrow \&_u[t_1,t_2]\ Q(X))$$

The second model, also discussed in [CLIF83], is the *continuity assumption.* It applies to cases when only one value of some semantic domain may qualify any given object at any one moment in time, i.e.,

$$\begin{aligned}&(\text{for-all } T,Y_1,Y_2,Q,X)\text{: } (\&T\ Q(Y_1)(X) \hat{} \&T\ Q(Y_2)(X) \hat{} \\ &\qquad\qquad \text{not}(Y_1 = Y_2) \rightarrow \text{ERROR}(\text{conflict},Q,X,Y_1,Y_2).\end{aligned}$$

This allows one to assume that a fact holds as long as no conflicting fact has been asserted:

$$\begin{aligned}&(\text{for-all } T_1, T_2, Q, Y_1, Y_2, X)\text{:}\\ &\quad (\&T_1\ Q(Y_1)(X) \hat{} \&T_2\ Q(Y_2)(X) \hat{} T_1 < T_2 \hat{}\\ &\quad (\text{for-all } T, Y)\text{:}\\ &\qquad (\&T\ Q(Y)(X) \hat{} T_1 < T < T_2 \rightarrow Y = Y_1) \rightarrow\\ &\qquad\quad \&u[T_1, T_2)\ m`Q(Y_1)(X))\end{aligned}$$

3.5.5.3. Cyclic phenomena

Before concluding the discussion on temporal qualification of facts, one last extension of the interval uniform operator is proposed. It is designed to facilitate the statement of facts which hold true at regular intervals of time. Daily tides, changes in the polar caps and seasonal vegetation changes are several phenomena characterized by regular cycles whose definition would benefit from the proposed extension.

The redefined interval uniform operator takes the general form

$\&_u[t_1,t_2](t_3,t_4)$ q(x)

where

t_1 is the beginning of the first cycle
t_2 is the end of the first cycle
t_3 is the beginning of the second cycle
t_4 is the upper bound for the end of the last cycle (optional)
$(t_2=t_1)$ is the length of each cycle
$(t_3=t_1)$ is the repetition period
$[t_1,t_2]$ may be open or closed at either end.

The notational convenience provided by this definition is clearly evidenced by the formulation of a statement such as "channel x has been closed every winter since 1970"

$\&_u$[dec-1970,feb-1971](dec-1971,now) frozen_over(x).

The discussion on temporal qualification of facts ends here. Although treated independently, temporal operators may be used in conjunction with the spatial operators defined so far and other operators introduced in sections to come. Operator commutativity may be assumed at all times.

3.5.6. Accuracy Qualification of Facts

The world about which GDP systems maintain information is full of uncertainties and the means by which the information is gathered are imperfect. Consequently, much of the information a GDP system provides to its users ought to be qualified in a manner that indicates the extent to which the information may be viewed as accurate. If this is not done, decisions taken under the assumption that the information is absolutely true may have disastrous consequences.

To qualify the accuracy of the information maintained by a GDP system presupposes the ability to specify the uncertainly level of some facts and the ability to evaluate the impact of logical inference on the accuracy of facts derived from them. This section shows how fuzzy logic may be used to accomplish these two goals. The presentation starts with a brief review of fuzzy logic. It is followed by the definition of a fuzzy operator and illustrations of how the operator allows one to specify uncertainty originating from five different sources: user confidence, measurement error, extrapolation, statistical sampling and non-determinism. The presentation then turns to a brief discussion of fuzzy constraints. The section concludes with the definition of a simple fuzzy inference model used for uncertainty level derivation in virtual facts.

3.5.6.1. Fuzzy logic

In two-valued logic the truth value of a formula may be 1 or 0, i.e., true or false. Fuzzy logic [LEE72], however, allows the truth value of a formula to take any value in the closed interval [0, 1]. The rules by which a truth value is assigned to a formula are modified accordingly.

The table below summarizes the truth value assignments corresponding to one of the most widely used rules, the min-max rule.

TRUTH(F) = TRUTH(q)	if F = q and q is an atomic formula
TRUTH(F) = 1 − TRUTH(F_1)	if F = ~F_1
TRUTH(F) = min(TRUTH(F_1),TRUTH(F_2))	if F = F_1 ^ F_2
TRUTH(F) = max(TRUTH(F_1),TRUTH(F_2))	if F = F_1 ¦ F_2
TRUTH(F) = inf{TRUTH(F_1(X)) for X in D}	if F = (for-all X)· (F_1(X))
TRUTH(F) = sup{TRUTH(F_1(X)) for X in D}	if F = (there-exist X): (F_1(X))

Using the min-max rule above, the sentence

flooded(plain) ^ frozen(plain)

is assigned

the truth value min(0.45,0.65), i.e., 0.45
when flooded(plain) has the truth value 0.45
frozen(plain) has the truth value 0.65

the truth value min(0,1), i.e., false
when flooded(plain) is false
frozen(plain) is true

The min-max rule is not the only rule that may be used in fuzzy logic and, like all the others, is limited in the extent to which it is able to combine symbolic logic and probability theory. In particular, it ignores possible logical dependencies between facts. Nevertheless, fuzzy logic provides an elegant and often intuitive way of assigning a measure of accuracy to information. The compatibility with two-valued logic (which may be seen as a special case of fuzzy logic) and with its inference mechanisms is also very attractive.

3.5.6.2. Uncertainty level specification

In general, there is no objective way of assigning an accuracy level to an arbitrary fact. While general models may be used to establish the propagation of the accuracy through the inference process, an initial set of accuracies must be provided by the user. The accuracy supplied by the user may vary in the degree of subjectivity depending on the uncertainty source affecting a particular fact. There are five main sources of uncertainty: user confidence, measurement error, extrapolation, statistical sampling, and non-determinism.

Before illustrating the way to specify the uncertainty level originating in each of the above sources, we will introduce a logical operator called the *simple fuzzy operator*. Its syntax takes the form "%a q(x)" where "a" is an accuracy value in the closed interval [0,1], zero is interpreted as absolutely false, one is interpreted as absolutely true and the values in between correspond to degrees of truth. The formal semantic definition is captured by the meta-fact

$$\text{(for-all } A,Q,X\text{)}: (\%A\ Q(X) \wedge (0 \leq A \leq 1) \rightarrow Q(A,X)$$

which states that accuracy is just another qualification of a fact.

For reasons dealing with the separation between accuracy and other concerns, the fuzzy operator may not qualify a basic fact. Instead, a separate virtual fact must be specified as in the definition

$$q(x) \rightarrow \%a\ q(x)$$

This will allow one to use different models of accuracy or to ignore accuracy all together.

The fuzzy operator allows the user to define rules by which the accuracy of certain classes of facts is determined. When accuracy is an expression of the user's trust in the data or in the measuring device, it may be derived from the fact in question. In other words, the accuracy becomes a function of the predicate, semantic domain values and the objects involved:

$$\text{(for-all } Q,Y,X,A\text{)}: (Q(Y)(X) \wedge (A = f(Q,Y,X)) \rightarrow \%A\ Q(Y)(X)).$$

The function f may be viewed, in some cases, as a model for estimating the precision of a measurement or sensing device and, in other cases, as a human judgment based on expertise that is not quantifiable. Any changes in the way accuracy is computed are limited to the redefinition of the function f.

Even when measurements are precise, uncertainty may still be introduced when extrapolations are made to fill in gaps in the data. Consider, for instance, a geological survey or an ocean temperature study. In both cases, a discrete set of points are sampled and the value attached to the points in between is computed using some mathematical formula. Given the frequency of the samples and some knowledge of the application, the extent to which the computed values differ from the real world may be determined and used as an accuracy estimate. The formula below is representative of the general form such accuracy definitions take:

$$\begin{aligned}
&\text{(for-all } A,P,P_1,P_2,Y,Y_1,Y_2\text{)}: \\
&\qquad (@P_1\ \text{temp}(Y_1)(\text{ocean}) \wedge @P_2\ \text{temp}(Y_2)(\text{ocean}) \wedge \\
&\qquad\ \text{neighbors}(P_1,P,P_2) \wedge \\
&\qquad\ ((A,Y) = f(P,P_1,P_2,Y_1,Y_2))) \rightarrow \%A\ @P\ \text{temp}(Y)(\text{ocean}))
\end{aligned}$$

where f is the temperature interpolation function.

If the user wishes to define accuracy as a statistical property of some group of sample facts (e.g., picture clarity may be expressed as one minus the percentage of cloud cover), any formalism based on pure logic fails. The reason is the inability to count the number of provable instantiations of some given formula. There is no way, for instance, to get a count of the number of pixels P that are white, i.e., render provable the formula "@P white(image)" where P is a free variable. To do this one needs to go outside pure logic and introduce a primitive such as

card("F(X_i)")

where F is a formula having the free variables X_i and "card" (read "cardinality") is a function that returns a count of the distinct instances of F(X_i) that are provable, if the number of instances is finite. For instance, if the only white pixels are p_1, p_2 and p_3, card("@P white(image)") returns the value 3. Moreover, picture clarity could be defined by the formula:

(for-all A): ((n = card("@P white(image)")) ^
(n_0 = card("@P any_color(image)")) ^
(A = 1 − n/n_0) → %A clarity(image))

The same paradigm may be actually used for any statistically defined accuracy.

A somewhat more exotic source of uncertainty is the introduction of virtual fact definitions which in effect represent non-deterministic processes that occur in time or space. Take for instance a very simple model for the propagation of a ocean wave along some direction that passes through the epicenter of the earthquake that produced the wave:

(for-all T_1,T_2,P_1,P_2,A_1,A_2,Y_1,Y_2):
(%A_1 @T_1 &P_1 speed(Y_1)(wave) ^
($T_2 = T_1 + t_0$) ^ ($P_2 = P_1 + t_0{*}Y_1$) ^
($Y_2 = Y_1/k_y$) ^ ($A_2 = A_1/k_a$) → %A_2 @T_2 &P_2 speed(Y_2)(wave))

where the constants k_y and k_a are measures of the decrease in the speed and the size of the wave, respectively.

We turn our attention next to some of the pragmatics of uncertainty level specification. There are three important points to consider. First, in those situations when accuracy qualifications are not relevant one should not have to contend with potential logical consequences of their unwanted presence. Second, it is possible for the same fact to be qualified in two different ways thus raising the issue of selecting the "right" accuracy. Third, the lack of a general accuracy model applicable to all facts suggests that alternate models are needed when the same data is used in different contexts.

There are two ways of ignoring accuracy qualifications: the user is interested only in the facts that are absolutely true or the user choses to view as true any facts whose accuracy exceeds certain threshold. In the first case, any definition supplied by the user simply ignores the presence of the fuzzy operator. As a consequence, since a formula such as q(x) is not provable from facts of the form %a q(x), regardless of the values taken by a, all the facts for which an accuracy is specified are automatically ignored. In the second case, the user must suply a meta-model defining the threshold rule as in (for-all A,Q,X) : (%A Q(X) ^ (A > a_0) → m'Q(X)).

A model must be specified in order to separate the facts of interest from all the other facts. Otherwise the rule above would have absolutely no effect since, usually, each fact for which an accuracy is specified also exists without any accuracy.

Because it is possible for several uncertainty level definitions to qualify the same fact as having several different accuracies, the ***unified fuzzy operator*** is introduced as a way of resolving such conflicts. This operator is restricted to appearing only to the left hand side of fact definitions and provides a way of refering to the highest accuracy assigned to some fact. For instance, to state that a fact is considered true whenever there is at least one accuracy qualification of this fact exceeding the value 0.75 one may write

(for-all A,Q,X) : (%[A] Q(X) ^ (A > 0.75) → m'Q(X)).

(Note: other definitions of the unified fuzzy operator, e.g., minimum or average value, may be needed for specific types of facts.)

Depending on the context, it is very likely that one may want to treat accuracy differently. In one case, for instance, 0.75 may be accepted as truth while in other situations 0.90 may be required. Each definition may be associated with a different model which is made part of the world view only when needed:

(for-all A,Q,X) : (%[A] Q(X) ^ (A > 0.75) → m_1'Q(X)).
(for-all A,Q,X) : (%[A] Q(X) ^ (A > 0.90) → m_2'Q(X)).

3.5.6.3. Fuzzy constraints

Any constraint that explicitly involves a fuzzy operator is called a fuzzy constraint. Two special cases are particularly relevant. The first one is when the error is not qualified by an accuracy but is triggered by the accuracy of some other fact as in the definition

(for-all A,X): (%A clarity(X) ^ (A < 0.80) → ERROR(bad_image,X)).

The second case is when some accuracy is actually associated with the error. It might be important, for instance, to know the percentage of river crossings where no bridge appears to be present:

(for-all A,P,N,N_0):
  ((N_0 = card("@P road(X) ^ @P river(Y)")) ^ not (N_0 = 0) ^
   (N = card("@P road(X) ^ @P river(Y) ^ @P bridge(X,Y)")) ^
   (A = 1 − N/N_0) ^ not(A = 0)) → %A ERROR(missing_bridge)).

A high accuracy value associated with this error may indicate possible problems with the data being processed.

3.5.6.4. Uncertainty level propagation via logical inference

In a previous section we have shown how an uncertainty level may be assigned to certain classes of facts. These facts in turn may be used to derive new virtual facts whose uncertainty level is dependent upon the accuracy of the original facts. The question addressed in this section is how to assign automatically an accuracy to facts derived from accuracy qualified facts.

We assume that each fact has a definition which does not involve any references to the accuracies of the facts involved. (We have adopted this approach earlier in order to separate the accuracy issues from the other issues and in order to permit easy substitution of one accuracy model for another.) Let us also assume the existence of a function AC which computes an accuracy for valid formulas. Since virtual fact definitions take the form

$$\text{(for-all } X_i\text{): } (F(X_i) \rightarrow q(X_k))$$
$$\text{with } i \text{ in } I,\ k \text{ in } K, \text{ and } K \text{ subset of } I,$$

the accuracy for $q(x_k)$ (i.e., an instance of $q(X_k)$) is given by $AC(F(x_i))$. This may be stated generally as

$$\text{(for-all } X_i\text{): } (F(X_i) \;\hat{}\; (A = AC(F(X_i))) \rightarrow \%A\ q(X_k))$$

(Note: These types of formulas may be generated mechanically.) Equivalently, the following meta-fact could be used:

$$\text{(for-all } X_k\text{): } (q(X_k) \;\hat{}\; (A = ACC(DEF(q),X_k)) \rightarrow \%A\ q(X_k))$$

where $ACC(DEF(q),x_k) = AC(F(x_i))$ for any x_i used to generate $q(x_k)$ and $DEF(q)$ returns the definition for q.

The definition of AC follows the basic rules of fuzzy logic introduced earlier

$AC(F(x_i))$

$= a$ when $\%[a]\ q_1(x_i)$ is provable
$=$ failure when $\%[a]\ q_1(x_i)$ is not provable
if $F(x_i) = q_1(x_i)$ with q_1 in PR and i in I;

$= \min(AC(F_1(x_{i1})),AC(F_2(x_{i2})))$
if $F(x_i) = (F_1(x_{i1}) \;\hat{}\; F_2(x_{i2}))$
with i1 in I1, i2 in I2, and I1 U I2 = I;

$= \max(AC(F_1(x_{i1})),AC(F_2(x_{i2})))$
if $F(x_i) = (F_1(x_{i1}) \mid F_2(x_{i2}))$
with i1 in I1, i2 in I2, and I1 U I2 = I;

$$
\begin{aligned}
&= \min(AC(F_1(x_i)),\\
&\qquad \inf\{\max(1 - AC(F_2(x_{i2},X_j)),\\
&\qquad\qquad AC(F3(x_{i3},X_j))) \text{ for all } X_j\})\\
&\quad \text{if } F(x_i) = (F_1(x_i)\ \hat{}\ (\text{for-all } X_j)\text{: } (F_2(x_{i2},X_j) \rightarrow F3(x_{i3},X_j)))\\
&\quad \text{with i2 in I2, i3 in I3, I2 U I3 subset of I,}\\
&\qquad\quad \text{j in J, and j not in I;}
\end{aligned}
$$

$$
\begin{aligned}
&= \min(AC(F_1(x_i)),1) && \text{when } F_2(x_{i2}) \text{ is not provable}\\
&= \text{failure} && \text{when } F_2(x_{i2}) \text{ is provable}\\
&\quad \text{if } F(x_i) = (F_1(x_i)\ \hat{}\ \text{not}(F_2(x_{i2})))\\
&\quad \text{with i2 in I2, and I2 subset of I.}
\end{aligned}
$$

Because of the reliance on fuzzy logic the approach described here inherits many of its limitations. The most important one is the inability to handle dependencies between facts. The conservative manner in which accuracies are computed does guarantee, however, that no fact will be given an accuracy greater than the one that would result from considering fact dependencies. Furthermore, if the only two accuracies used are 0 (false) and 1 (true) and the facts involved are independent the results are consistent with the two-valued logic, e.g., when "$q_1\ \hat{}\ q_2 \rightarrow q_3$" the accuracy of q_3 is 1 if the accuracies of both q_1 and q_2 are 1 and is 0 if q_1 or q_2 have an accuracy of 0. If either q_1 or q_2 have no accuracy assigned then q_3 will have no accuracy also.

3.5.7. Security Qualification of Facts

This section is concerned with ways of specifying security needs in the confines of the proposed formalization of geographic data processing requirements. Security is important to any system, military or civilian, and it involves a broad range of issues. We consider here only the issue of unauthorized access to facts maintained by the system. In a national security sensitive system the security requirements discussed here would relate mostly to the enforcement of need-to-know rules rather than to the support of multiple levels of security. The reason is not so much theoretical but pragmatic.

The discussion is divided into three parts. First, we present several methods for specifying the security requirements of various types of facts. Second, we review the concept of security integrity and show how to express it using our formalization. We also consider the issue of assigning a classification to facts inferred from facts for which a security classification has been specified. We conclude by identifying several important open issues in the security requirements area.

3.5.7.1. Security requirements specification

The sensitivity of an individual fact is indicated by the classification assigned to it. For simplicity sake, it is assumed the set of possible classifications is finite and is ordered from low to high. (The total ordering is not actually necessary and the definition of need-to-know requirements is better captured by a partial ordering. The extension from a total to a partial ordering, however, is straightforward.) SC (read "security classifications") is used to denote the characteristic function of the domain of classifications. The only primitive operations permitted over classifications deal with comparing them against each other.

The *simple security operator* is introduced as the means for associating a security classification to a fact. Its syntax takes the form "$\$c\ q(x)$" where "c" is a valid security classification. The formal semantic definition is captured by the meta-fact

$$(\text{for-all } C,Q,X)\text{: } (\$C\ Q(X) \wedge SC(C) \rightarrow Q(C,X)$$

which states that the security classification is just another qualification of a fact.

For reasons dealing with the separation between security and other concerns, the security operator may not qualify a basic fact. Instead, a separate virtual fact must be specified as in the definition

$$q(x) \rightarrow \$c\ q(x)$$

This will allow one to use different models of security without altering the rest of the requirements.

The security requirements for a GDP system consist of rules that establish the way in which classifications are assigned to individual facts. These rules may assume a variety of forms:

(1) by object (e.g., port)—when the same classification is assigned to any fact about a particular object, unless several objects are referenced in some fact. In this case the fact receives the highest classification associated with any of objects involved;

$$\begin{aligned}&(\text{for-all } Q,Y_i,X_j,C)\text{: } (Q(Y_i)(X_j) \wedge \\ &\qquad (C = \sup\{S\_REQ(X_j) \text{ for all } i\}) \rightarrow \$C\ Q(Y_i)(X_j))\end{aligned}$$

(2) by predicate (e.g., depth)—when the same classification is assigned to any fact involving the same predicate;

$$\begin{aligned}&(\text{for-all } Q,Y_i,X_j,C)\text{: } (Q(Y_i)(X_j) \wedge \\ &\qquad (C = S\_REQ(Q)) \rightarrow \$C\ Q(Y_i)(X_j))\end{aligned}$$

(3) by value (e.g., depths less than 50 meters)—when the classification is assigned based on some property of the semantic domain value appearing in the fact;

$$\begin{aligned}&(\text{for-all } Q,Y_i,X_j,C)\text{: } (Q(Y_i)(X_j) \wedge \\ &\qquad (C = S\_REQ(Q,Y_i)) \rightarrow \$C\ Q(Y_i)(X_j))\end{aligned}$$

(4) by object/predicate (e.g., depth of certain ports)—when the classification is assigned based on the predicate and the objects involved;

$$(\text{for-all } Q,Y_i,X_j,C)\text{: } (Q(Y_i)(X_j)\ \hat{}\ (C = S\_REQ(Q,X_j)) \rightarrow \$C\ Q(Y_i)(X_j))$$

(5) by fact (e.g., certain depths of certain ports)—when the classification is assigned based on the particulars of each fact;

$$(\text{for-all } Q,Y_i,X_j,C)\text{: } (Q(Y_i)(X_j)\ \hat{}\ (C = S\_REQ(Q,Y_i,X_j)) \rightarrow \$C\ Q(Y_i)(X_j))$$

When several types of rules are used together, it becomes possible for a fact to be assigned several classifications. The *unified security operator* is introduced as a way of resolving such conflicts. This operator is restricted to appearing only to the left hand side of fact definitions and provides a way of refering to the highest classification assigned to some fact, also called the security level of that fact. Let us assume, for instance, that each model M in the system has a particular classification associated with it (denoted by CLEARANCE(M)) and that no facts having a security level that exceeds M's clearance may be true in M. This could be stated by employing the unified security operator:

$$(\text{for-all } M,C,Q,X) : (\$[C]\ Q(X)\ \hat{}\ (C \leq CLEARANCE(M)) \rightarrow M'Q(X)).$$

3.5.7.2. Security requirements integrity

The manner in which security requirements are defined above does not take into consideration dependencies between facts. Consequently, secure facts may be leaked although the user has no direct access to them. Take, for instance, the following definitions

$$(\text{for-all } X)\text{: } (q_1(X) \rightarrow q_2(X))$$
$$(\text{for-all } X)\text{: } (q_1(X) \rightarrow \$c_1\ q_1(X))$$
$$(\text{for-all } X)\text{: } (q_2(X) \rightarrow \$c_2\ q_2(X)).$$

If q_1 is less classified than q_2, i.e., $c_1 < c_2$, by knowing q_1 and the first of the three formulas a user may easily deduce q_2 even when the classification of q_2 exceeds the user's clearance.

Any assignment of security classifications that permits the leaking of sensitive data is said to violate the integrity of the security requirements. What is meant by a secure (i.e., leak free) system, however, differes from one circumstance to another. Many definitions have been proposed. They take the form of security models [LAND81]. Among them, the Bell and LaPadula model appears to be most appropriate for use in our context. Their model assumes that users have available four modes of access to the files of the system: execute-only, read-only, append, and read-write. A system state is said to be secure if it satisfies the following conditions:

(1) no user may have append access to a file whose security is smaller than the user's clearance;

(2) no user may have write access to a file whose security level is different from the user's clearance;

(3) no user may have read access to a file whose security level is greater than the user's clearance.

The use of the Bell and LaPadula model in conjunction with our formalization is complicated by differences of context. First, we have no files but facts and formulas defining new facts from existing ones. Second, logical inference does not have a clean correspondent in their model. We were able to establish a clean correspondence for only two of the four operations assumed by Bell and LaPadula:

(1) execute-only corresponds to the use of a formula without knowledge of what it is, i.e., knowledge of a virtual fact but not of its definition;

(2) read-only corresponds to having knowledge of a formula or fact but also of any fact derivable from the accessible formulas and known facts;

(3) append and read-write correspond to the addition/deletion of facts and formulas but the latter operations are much more complicated because the potential side effects on the inference process.

We are now in the position to illustrate the integrity constraints for the case when the addition and deletion of basic and virtual fact definitions are prohibited and when execute-only always implies read-only. In this restricted case, the integrity constraints may be stated as follows:

(1) no user has access to any fact or formula whose security level exceeds the user's clearance;

(2) no user may deduce any formula having a security level that exceeds the user's clearance;

(3) no user may infer any fact having a security level that exceeds the user's clearance;

(4) no user may infer any fact derived from a fact having a security level that exceeds the user's clearance;

The first constraint requires the existence of a procedure by which to determine if a user having a clearance c_0 may be given access to individual formulas or facts. If, in addition to the unique security level of each fact (i.e., $[c]q(x)), we introduce a unique security level for its definitions (e.g., DEFSEC(q)), the procedure is reduced to a mere test. The actual mechanism by which the segregation between what is and what is not accessible is not important at this point.

The remaining constraints may be satisfied if the security level of each fact equals the highest security level of any fact or formula involved in its derivation. The following example is used to show the types of problems that may arise if this condition is not satisfied.

(for-all X): $(q_1(X) \rightarrow q_2(X))$ — security level c_{12}
(for-all X): $(q_1(X) \rightarrow \$c_1\ q_1(X))$ — security level c_1
(for-all X): $(q_2(X) \rightarrow \$c_2\ q_2(X))$ — security level c_2
$c = MAX(c_1, c_{12})$
c_0 = the user clearance

When c is not equal to c_2 there are several cases to consider.

Case 1: if $(c_2 > c)$ a user having the clearance c_0=c is able to derive q_2 from q_1 and the first formula. This violates constraint 3.

Case 2: if $(c_1 \geq c_{12})$ and $(c_2 < c)$ a user having the clearance $c_0=c_2$ has access to a fact derived from a fact having a security level higher than the user's clearance. This violates constraint 4.

Case 3: if $(c_{12} > c_2)$ and $(c_2 < c)$ a user having the clearance $c_0=MAX(c_1,c_2)$ may deduce the definition of q_2 from the one-to-one correspondence between q_1 and q_2. This violates constraint 2.

The meta-constraint stating the condition above takes the form:

(for-all C,Q,Y,X): ($[C] Q(Y)(X) ^
(C < SEC(DEFSEC(Q),DEF(Q),Y,X))
→ ERROR(security_violation,Q,Y,X))

The function SEC returns the highest security level encountered in generating Q(Y)(X). In the case of q_2, for instance, DEFSEC(q_2) is c_{12} and the only fact involved is q_1 which has the classification c_1. Consequently,

$$SEC(DEFSEC(q_2),DEF(q_2),X)) = MAX(c_{12},c_1)$$

The formal definition of SEC is similar to the one given for ACC in a previous section and will not be expanded here. The analogy with ACC also suggests to use SEC to generate the right security for virtual facts having no explicit security requirements but derived from facts for which a security level has been defined. The only meta-fact needed for this purpose is similar to the meta-constraint above:

(for-all C,Q,Y,X): (Q(Y)(X) ^
(C = SEC(DEFSEC(Q),DEF(Q),Y,X))
→ $C Q(Y)(X))

This establishes the propagation of security requirements due to the inference process. However, the approach works only if the virtual fact definitions have some security level associated with them or if some default is provided for cases when no security level has been supplied.

3.5.7.3. Open problems

Security requirements issues that remain unresolved fall into two categories: those that have been identified through the use of the logic based formalization and those which, although recognized by others, assume a different form in the context of our work.

The first category includes the problems relating to the addition and deletion of basic and virtual fact definitions. Current work on security ignores logical dependencies between objects. Consequently, the deletion of an unclassified file has no possible consequence on any of the classified ones. On the contrary, in our formalization, the deletion of a basic fact may lead to the immediate deletion and/or creation of any number of facts having higher levels of security. This suggests that the logic model we have adopted is more general than the models considered so far and, therefore, a better suited relm for the study of security issues.

The second category includes, among others, the issue of desensitizing information. Let us assume, for instance, a case where the position of two targets is sensitive but the distance between them is not. According to the integrity constraints defined in this section, however, the distance which is derived from the positions must have the same security level as they have. The solution that is being advocated is the introduction of so-called *trusted formulas* (subjects or processes in the terminology used by other authors). Sufficient criteria for guaranteeing trustworthiness, however, still have to be developed.

3.5.8. Conclusions

The formalization introduced in this section provides a conclusive demonstration of the feasibility of expressing in logic complex geographic data processing concepts and ways of reasoning about them. The use of logic adds power and flexibility to the process of specifying and validating GDP requirements by enabling one to explicitly state and evaluate both data and knowledge requirements within the context of a single unified formal foundation. Moreover, the approach has the potential to lead toward the development of a requirements engineering environment that allows one to actually reason about requirements rather than merely state them.

3.5.9. References

[CLIF83] Clifford, J. and Warren, S., "Formal Semantics for Time in Databases," *ACM Trans. on Database Systems* 8, No. 2, pp. 214-254, June 1983.

[LAND81] Landwehr, C. E., "Formal Models for Computer Security," *Computing Surveys* 13, No. 3, pp. 247-278, September 1981.

[LEE72] Lee, R. C. T., "Fuzzy Logic and the Resolution Principle," *JACM* 19, No. 1, pp. 109-119, January 1972.

[MINK78] Minker, J., "An Experimental Relational Data Base System Based on Logic," pp. 107-147, *Logic and Data Bases,* H. Gallaire and J. Minker editors, Plenum Press, 1978.

[NAGY79] Nagy, G. and Wagle, S., "Geographic Data Processing," *Computing Surveys* 11, No. 1, pp. 139-181, June 1979.

[RESC71] Rescher, N. and Urquhart, A., *Temporal Logic,* Springer-Verlag, 1971.

[SILV77] Silver, J., "The GBF/DIME System: development, design and use," U.S. Bureau of the Census, Washington, D.C., 1977.

[ZOBR81] Zobrist, A. L. and Nagy, G., "Pictorial Information Processing of Landsat Data for Geographic Analysis," *Computer* 14, No. 11, pp. 34-41, November 1981.

4. TECHNOLOGY EVALUATION METHODS

4.1. INTRODUCTION

In Section 3 we introduced a requirements specification and validation methodology. Its relevance to technology evaluation is captured by two important attributes. First, it makes explicit the impact of technology on the production environment. Second, it allows one to focus the scope of technology studies on the needs of a particular class of production problems and on a specific domain of design solutions. This section further explores some of the technical details involved in exploiting the latter aspect of our methodology.

It is our contention that making policy decisions based on the results of technology studies whose context is not bound by specific classes of problems and designs involves unacceptable risks. In describing the requirements validation methodology we have shown how to maintain requirements visibility throughout design and technology evaluation studies and how to use the results of these studies to estimate the production impact of proposed systems and/or technologies. We turn now to the issue of how to carry out the design and technology studies and what kind of technical support should be made available to the designer performing such investigations. Short range concerns are addressed by developing a system design methodology aimed at facilitating the interaction between requirements validation and technology evaluation. Long range concerns are addressed by identifying the kind of tools needed to support the design and technology studies. These studies supply the performance data required by the realistic simulations used in requirements validation.

Section 4.2 provides an overview of the Total System Design (TSD) Framework which laid the technical and philosophical foundation for this work. Of particular interest is the framework's attempt to formalize the role played by technology in the design process.

Section 4.3 describes an integrated approach to design and technology evaluation. Fundamental to our approach is the reliance on system level models that enable the designer to formulate and answer questions regarding the system's logical and performance characteristics when the interaction between the hardware and the software is important, i.e., when the impact of faults, failures, communication delay, hardware selection, scheduling policies, etc. must be considered. In the simplest terms, our concern extends beyond the traditional (isolated) software and hardware concerns by addressing the issue of software correctness and performance characteristics when running on a particular distributed hardware architecture and using a particular operating system.

Section 4.4 describes the technical foundation for the methodology introduced in Section 4.3. The models we use are called *Virtual Systems* and represent either abstractions of existing systems or definitions of proposed systems. A virtual system consists of six components of a distributed system. The *Functionality* is an abstraction of the processes which carry out the system function (e.g. the applications software).

The *Architecture* captures the overall hardware organization and distribution of the system. The *Scheduler* together with the *Allocation* define the relationship between functionality and architecture, and the changes which the relationship undergoes with time. *Allocation,* while related to the notion of partition present elsewhere, is much more general. It covers static and dynamic allocation of functions (in the functionality) to processors (in the architecture) and the allocation of time and space on an individual processor to the functions associated with it. The *Performance Specification* is an abstraction of both workload characteristics (the environment model is an integral part of the virtual system) and measurement probes. The performance specification may be used to explicitly state the assumptions made by designers regarding the characteristics of the environment and of the system components to be utilized in the realization of the system. The performance specification is coordinated with the rest of the model via the *Instrumentation*. Related to Section 4.4 are Appendices C and D.

We experimented with specifying virtual systems using a language called CSPS *(Communicating Sequential Processes with Synchronization)*. Appendix D includes a set of system models built using CSPS. The application software is assumed to involve the selection of an acceptable road path between two points using elevation data about the region of interest. Several architectures are considered: uni-processor, dual-processors with point to point communication, dual-processors with bus communication and dual-processors with point to point communication and dynamic reallocation. Appendix C provides a formal definition for the CSPS syntax used in this report.

4.2. TSD PHILOSOPHY

The modern computer-based system is a complex structure, containing many kinds of functionality. At the highest level, it is a network of interacting processes that provide the application-oriented functions of the system. This processing network is supported by a hardware architecture and (usually) an executive system. The executive functions and the application functions are implemented by software and (sometimes) firmware. In some cases, special hardware functions are needed, and they must be designed.

Successful development of computer-based systems requires highly integrated design methodologies covering the entire system development life-cycle from problem definition to final testing and spanning across the traditional hardware/software boundaries. Furthermore, to be effective they must be specialized for a particular application, technology and organization. The application forces the methodology to address issues such as response time requirements in real-time systems, the need for formal correctness proving in the security kernels of information systems, and special test procedures for systems where "fixes" during system deployment are not possible (e.g., spacecraft, satellites, certain high-effort weapons, etc.). Methodologies exploit available technology with the aim of enhancing both design productivity and quality. Program generators and microprocessor based architectures, for instance, serve this purpose in business data processing and industrial control, respectively. Finally, the utility of a methodology is severely limited unless it takes into consideration corporate resources such as available manpower, the training and experience of personnel, available support tools, and project management practices.

The Total System Design (TSD) Framework[1] supports the development of integrated system design methodologies by providing a technical framework that identifies and structures the issues that must be addressed during system development. The purpose of the framework is to aid in our understanding of the design process, to serve as a foundation for the development of new methodologies and to provide a means for methodology analysis and selection. Furthermore, its high degree of generality and broad scope make the TSD Framework useful in the establishment of company wide system development standards, even for organizations whose concerns span several application areas. The framework is compatible with Department of Defense (DoD) standards and with research results described in the professional literature.

The TSD Framework partitions the system development life-cycle into stages and phases. The stages constitute a natural structuring based on major differences in applied technology, while the phases that make up a stage impose an ordered, layered approach to design, thus reducing the risk of error and producing systems that are easier to understand and maintain. Associated with each phase there is a set of design activities, henceforth called steps, that are needed to carry out the design process within a phase. The goal of these steps is to facilitate the formal documentation and evaluation of the evolving design. The techniques required to perform each step, however, are not part of the framework. This is because they can differ from phase to

[1] G.-C. Roman et al. "A Total System Design Framework," *Computer* 17, No. 5, pp. 15-26, May 1984

phase in the same methodology and from one methodology to another.

The particular stages, phases and steps described in this report represent a specific "state" of the framework. This state captures the current understanding of the system design process, and, since this body of knowledge is still evolving, the framework is also evolving. *The current state reflects a growing understanding of hardware/software trade-offs and gives formal recognition to the major influence of technological considerations on the design process.*

The framework forces a rational treatment of hardware/software trade-offs by imposing a top level system architecture design phase in which the system is considered a network of interacting concurrent processes. Hardware and software considerations are allowed to influence this processing model, but only in a general sense, such as the number of processors that might be needed to meet workload estimates, or the number that might be required to provide for fault tolerance or graceful degradation. The final binding to specific hardware and software happens at a subsequent phase. This arrangement ensures that the choice of hardware and software is based on a well considered view of the processing needs of the system.

A formal design activity within each phase (the inference step) forces a review of the implications of current design decisions on the hardware technology and computational techniques that subsequent phases can use to support these decisions. This allows a top-down approach to design with minimum risk of having to backtrack, and at the same time, requires that the capabilities of current technology be reviewed from both the computational and cost standpoints.

Our overview of the TSD Framework begins with a description of its stages, phases, and steps. This is followed by a discussion of the hardware/software trade-offs issue and a summary of the framework's relevance to technology assessment.

4.2.1. Stages and Phases

This section describes the stages and phases of the TSD Framework, the rationale behind its structure and the features that distinguish it from other views of the system development life-cycle.

PROBLEM DEFINITION STAGE. This phase determines the functional and the non-functional requirements that must be met by the computer-based system. Its presence in the framework underscores the idea that successful system design must start with a clear understanding of the problem being addressed. This task is accomplished by an identification phase and a conceptualization phase.

Both phases are application domain dependent and their successful completion rests on a good understanding of the application. The IDENTIFICATION phase is informal in nature and has an exploratory flavor. Its objective is to produce an identification report which contains all the information available with regard to the system responsibilities, system interfaces, and design constraints. Despite the fact that the level of formalization and abstraction of the identification report is relatively low, the report serves two important functions: it establishes the communication link between the designer and the user and provides the necessary base for the development of a formal definition of the problem. This formal development is done in the conceptualization phase.

The CONCEPTUALIZATION phase uses the identification report to generate the system requirements. These requirements contain a conceptual model which formalizes the system's role from a user perspective and the design constraints imposed by the application. The dataflow diagrams used in system analysis and the relational model used as a database specification are examples of two conceptual models embodying different levels of formality and addressing different types of problems. Because of its formal nature, the conceptual model provides a solid basis for the entire design process and represents the ultimate correctness criterion against which the final system functionality is judged. Another fundamental acceptance criterion is the ability to meet all the stated constraints.

SYSTEM DESIGN STAGE. This stage bears the responsibility for establishing the hardware and software requirements for each component of the system. They consist of functional and non-functional capabilities of the components and the interfaces among them (e.g., communication protocol, programming language support, operating system primitives, etc.). These requirements are used as the basis for the procurement and/or design of the individual system components (e.g., computers, operating systems, packages, etc.) and their subsequent integration into a delivered system.

The system design stage includes two phases: system architecture design and system binding. The main concern of the SYSTEM ARCHITECTURE DESIGN phase is to investigate system design alternatives and their potential impact on the choices for a feasible system configuration (i.e., hardware/software mix). In the case of a distributed database system, for instance, this phase may identify the data and processing distribution, the number of nodes present in the network and their general characteristics but not any particular computer to be used at the nodes or any specific software packages. Therefore, without making any explicit choices with respect to the selection of particular software or hardware components, this phase is implicitly involved in the performance of hardware/software trade-offs to the extent that design decisions taken here determine the system-level components that must be supported by hardware and software.

It is the responsibility of the SYSTEM BINDING phase to select the actual mix of hardware and software. The hardware and software requirements being generated by this phase may assume a variety of hardware/software combinations from off-the-shelf complete systems to custom-built components. The election of one option over another is determined by the nature of the system design, the constraints to be met, and the available technology. Binding options are identified in the system architecture design phase, but the selection of specific components is done in the binding phase. The system designer may postulate, for instance, the use of minicomputers having certain capabilities, price range and software support. The actual selection, however, occurs in this phase which is also charged with specifying the requirements for the software that would still need to be produced.

SOFTWARE DESIGN STAGE. This stage includes all activities relating to software development and/or procurement. There are three phases involved in this stage. The first one, SOFTWARE CONFIGURATION DESIGN, is responsible for the procurement of off-the-shelf software as well as the overall high-level design of the software system for each programmable system component. By high-level software system design, we mean the structuring of the software into subsystems, virtual layers, tasks, etc., and the definition of the interfaces between such components, henceforth referred to as programs. The software requirements are the basis for these activities which result in the development of program requirements specifications. The PROGRAM DESIGN phase, in turn, takes these requirements and produces the program design (data and processing structures) which, together with all pertinent assumptions and constraints, make up the implementation requirements. The implementation requirements are used in the CODING phase to build the actual programs.

MACHINE DESIGN STAGE. This stage plays a role similar to that of the first two phases of the software design stage. The HARDWARE CONFIGURATION DESIGN phase is concerned with the procurement of off-the-shelf machines and the design of the high level architecture of custom hardware. Component requirements are developed for all entities that are part of the custom hardware and passed on to the COMPONENT DESIGN phase. This phase generates a register transfer level machine description which is used to determine the circuit design requirements and the firmware requirements.

CIRCUIT DESIGN STAGE. This stage follows a generally accepted scenario involving three phases: SWITCHING CIRCUIT DESIGN, ELECTRICAL CIRCUIT DESIGN, and SOLID STATE DESIGN. Each phase generates design requirements for the phase listed after it.

FIRMWARE DESIGN STAGE. This stage consists of three phases that are an analog to program design, coding, and compilation. These phases are called MICROCODE DESIGN, MICROPROGRAMMING and MICROCODE GENERATION.

When comparing the TSD Framework against other work and current DoD standards, the following features stand out:

(1) The generation of system requirements (i.e., problem definition) involves two phases, identification and conceptualization.

(2) System design is separated from both problem definition and software design.

(3) System design is split into two phases called system architecture and binding.

This last point is particularly relevant when investigating technology evaluation issues.

4.2.2. Steps

The previous section gave a general introduction to the design areas covered by the TSD stages and phases. This section gives a general introduction to the activities that occur during the design process. The major design activities within a phase are called STEPS. A step is not a design activity that takes place at a particular point in the design process but an abstraction over all activities that contribute to accomplishing the design objective encapsulated in the definition of the step. Consequently, the concept of step as used in this report is non-procedural in nature. This feature distinguishes a methodological framework from a methodology which is intrinsicly procedural in nature and allows one to employ the framework as a specification for a class of methodologies sharing the same design philosophy.

There are ten steps which collectively represent the activities within any phase, regardless of the nature of the phase.

The FORMALISM SELECTION step encompasses the activities involved in selecting a formalism for a particular problem domain. Each phase may involve one or more different formalisms: programming languages for coding; pseudocode for program design; logic diagrams for switching circuit design; stimulus-response graphs; mathematical and logic models for conceptualization; etc. Candidate formalisms are evaluated for their expressive power in that domain and also for qualities such as simplicity of use, lack of ambiguity, analyzability, and potential for automation. While this step must take place before other steps in the phase, it often occurs long before them. This is sometimes due to the use of a methodology that is based on a particular formalism, but is more often simply a matter of policy or is due to the availability of tools tailored to that formalism.

The FORMALISM VALIDATION step encompasses activities involved in determining whether a formalism has the expressive power needed for a particular task. It also includes the evaluation of formalisms from the standpoint of ease of use. These tasks are generally non-trivial and may involve both theoretical and experimental evaluations. Theoretical results may indicate the power and the fundamental limitations of the formalism while past experience with it on similar projects may provide insight into its appropriateness and ease of use. One project may reject FORTRAN because it does not support recursion while another may consider it a potential source of maintenance problems due to its limited support for structured programming. Finite state machines may be employed in defining some communication protocols but are inappropriate when unbounded queues are present and prove cumbersome when the number of states is large. The step also includes evaluations of the formalism's potential for design automation (as a way to bring about productivity increases) and its ability to support hierarchical specifications (as an aid to controlling complexity).

The EXPLORATION step encompasses the mental activities involved in synthesizing a design. These activities are creative in nature and depend on experience and natural talent. They cannot be formalized or automated unless the problem domain is restricted to a significant degree.

The ELABORATION step encompasses the activities involved in giving form to the ideas produced in the exploration step through the use of various formalisms. Coding, specification writing, and circuit layout drawing are typical activities associated with this step, but its scope extends to the building of a concrete object such as a piece of hardware. The effectiveness of this step may be greatly increased through the use of a variety of design and manufacturing aids.

The CONSISTENCY CHECKING step encompasses activities such as checking for incorrect uses of formalisms, checking for contradictions, conflicts, and incompleteness in specifications, and checking for errors of a semantic nature. This includes checking for consistency between different levels of abstraction in a hierarchical specification and the reconciliation of multiple viewpoints. The presence of a subroutine which is never called is one example of the type of problems addressed by consistency checking. Another is a mismatch between the number of input lines for two instances of the same device on some logic diagram.

The VERIFICATION step encompasses activities involved in demonstrating that a design has the functional properties called for in its requirements specification. Since each phase has a requirements specification and produces a design, this step is equally important for all phases. Proving program correctness is a common example of this type of activity. The difficulty of this task is representative of the difficulty of the verification task in general.

The EVALUATION step encompasses activities involved in determining whether a design meets a given set of constraints. This includes constraints which are part of the requirements specification for the phase and constraints which result from design decisions. The nature of the evaluation activities depends on the type of constraints to be analyzed. The evaluation activities include: classical system performance evaluation of response time and workload by means of analytical or simulation methods, deductive reasoning for investigating certain qualitative aspects like fault tolerance or survivability, and construction of predictive models for properties such as cost and reliability.

The INFERENCE step encompasses activities involved in assessing the potential impact of design decisions made in the phase. The domain of these activities includes: impact on the application environment (e.g., can we afford the weight? is the personnel retraining too expensive?), ability of subsequent phases to live with decisions made in this phase (e.g., is the bandwidth achievable?), effect on system maintainability and enhancability (e.g., will these parts be available five years from now?), and effect on implementation options (e.g., is there a good reason why the use of a mainframe is ruled out?). While these issues must be considered in every phase, proper treatment is particularly critical in those stages defining architectures.

The INVOCATION step encompasses the activities associated with releasing the results of the phase. This includes quality control activities where tangible products are involved and review activities leading to the formal release of output specifications. This latter aspect gives the step its name, since the release of specifications in effect invokes subsequent phases.

The INTEGRATION step encompasses the activities associated with the configuring and testing of that portion of the total system that was designed in the phase. Although it is traditional to consider integration to be a design area that would qualify as a stage in the framework, the integration activities have been distributed among the phases in recognition of the following facts: the expertise needed to test a portion of the system is similar to the expertise needed to create its requirements; the assumptions made in a phase about the nature of the products that could be delivered by the subsequent phases must be checked once the subsequent phases complete their respective tasks; all models used to make these assumptions must be validated; and, finally, errors found during integration must naturally be resolved in the phase that created the requirements.

4.2.3. Hardware/Software Trade-Offs

This topic, introduced earlier in the context of the system design stage, is fundamental to understanding the role technology plays in the design process. In this section we provide a brief overview of the framework's perspective on this issue. The discussion starts with the definition of hardware/software trade-offs, outlines the approach prescribed by the framework, and identifies the main problem areas.

The problem embodied in hardware/software trade-offs is that of allocating the system's functionality among hardware and software components in a manner that satisfies all system design constraints. This activity embraces more than just determining which functions are to be performed by hardware and which by software. First of all, hardware and software may be off-the-shelf, customized, or custom-made. In each case the cost and risk equations are different. Custom hardware may be faster but, even when it is reasonably priced, may face the designer with the possibility of greater risks and higher maintenance costs. A slower machine that comes with the right operating system and an adequate database management system is usually preferred over a faster and cheaper machine that lacks software support. For this reason, a system for which a lot of software is available may be selected even if it represents a higher investment option.

The system architecture design phase is engaged in a systematic process of reducing the binding options to the point where the binding phase is left to deal strictly with component selection from among a few feasible alternatives. Every system architecture design decision has implications with respect to the type of technology that would be needed to realize the system. Furthermore, partitioning into hardware and software needs to be carried out as part of this phase because all performance models used in the evaluation and inference steps demand, as a minimum, information about the distribution of the system's functions among various processors and about interprocessor communication costs. All such design decisions are actually subject to explicit review and analysis in the inference step. Of particular concern for the inference step is the rejection of any design solutions which unnecessarily limit the range of feasible binding options. Since the system architecture is presumed to be developed top-down, the option elimination process is characterized by an iterative sequence of refinements and inferences.

Having the range of binding options significantly reduced by the previous phase, system binding concentrates on selecting specific components among those still eligible. It is critical to proceed with the selection of individual components in the context of the entire system, and not by optimizing local decisions. This enables the focus to remain on the performance objectives of the system as a whole (cost included), where it belongs.

Neither option reduction nor component selection is a simple task. The former requires significant experience with system design and a good grasp of existing technology and current technological trends, issues which are difficult to formalize. The availability of appropriate performance models applicable both to performance evaluation and to technological inferences could, however, assist the designer in very important ways. While the number of conceivable binding options may be overwhelming, the development of reduction strategies and performance models for a few common ones is believed to be feasible, but nontrivial. Similar challenges are present in dealing with component selection in the binding phase. On the one hand, there is a need to develop adequate selection strategies for both software and hardware components. On the other hand, it is necessary to establish meaningful mappings between performance attributes present in the performance models mentioned above and those recognized in the actual component candidates.

4.2.4. Technology Evaluation Implications

The TSD Framework builds directly on the current understanding of system design methodologies with respect to both the phases and the steps that make up its structure. Its steps represent a taxonomy of the design activities generally encountered in system design. Its phases, aside from those included in the system design stage, have been recognized already by other authors. There are, however, two important distinctions between the way phases and steps are used here and elsewhere. First, the grouping of activities into a phase is based upon the nature of the technical expertise they require rather than upon considerations related to project management. The latter are relegated to methodologies and are not part of the framework. Second, the steps are abstractions over classes of design activities and not specific actions to be carried out by the designer in some prescribed order. These differences stem from the fundamental distinction between frameworks and methodologies.

One aspect of the TSD Framework that has not been exploited yet is its use as a technical foundation for systematic evaluation of technological trends and implications. *By formalizing the relationship between technology and system development, the TSD Framework makes possible meaningful technological evaluations.* In accordance with the TSD Framework, the system design stage is where the broader technological issues are being addressed and the dynamics of hardware/software trade-offs capture the essence of the interaction between system development and the postulated or available technology.

Because the system architecture design phase is involved in the exploration of the design space, its treatment of technological issues lays the foundation for a systematic approach to the investigation of long range technological trends and their impact on the production environment. Similarly, the system binding phase is relevant to the study of short range issues such as the evaluation and comparison of specific technological options. In the next section we will show how these considerations have led to using system design methods to derive meaningful systematic technology evaluation approaches.

4.3. INTEGRATED DESIGN/TECHNOLOGY EVALUATION

Technology includes methods for accomplishing technical tasks and the physical means by which a system can be implemented. This report distinguishes between two types of technological investigation: technology study and technology evaluation. A technology study looks at technological trends in terms of general developments in some particular area of technology, but independent of how this technology may actually be utilized. This definition differs from that of technology evaluation, which, given a particular state of technology, involves answering how that particular technology will affect a particular application. In other words, technology evaluation concerns how some technology that exists or is postulated will impact some application.

4.3.1. Motivation

Our objective is to establish a methodology for technology evaluation. This methodology is expected to improve the accuracy of technology evaluation, to make technology evaluation results useful for forecasting production capabilities, to reduce risks of design and planning of new systems, and to make it possible to predict both the long and short range impact of different types of technologies on production. A methodology of this sort will benefit those involved in research and development as well as those involved in R&D management and technology transfer efforts.

Our approach is rooted in the TSD philosophy and emphasizes the integration of system design, technology study, and requirements validation. The TSD framework is relevant because it formalizes the impact of technology on system development and indicates where technology has greatest impact in the development process, which is the system design stage. Consequently, system design methodologies rooted in the TSD philosophy hold the best promise for being the basis for technology evaluation.

Current technological trends and system design practices indicate that this objective, to establish a methodology for technology evaluation, is appropriate and necessary. Technology is changing rapidly; system developments exploit technology to its limits; system developments often rely on an ambiguous picture of technology; and often system developments are very ambitious undertakings, creating novel systems. A methodology that can increase confidence in current system developments, improve long range planning and upgrade potential, and lend a deeper understanding of qualitative implications of quantitative technological advances would be of great benefit particularly in the GDP area where the production is expected to be heavily dependent upon the performance of the systems being built today. The next section provides an overview of our proposed technology evaluation methodology.

4.3.2. Approach

Our technology evaluation strategy relies on the synergistic use three different methodologies. (Figure 4.3-1)

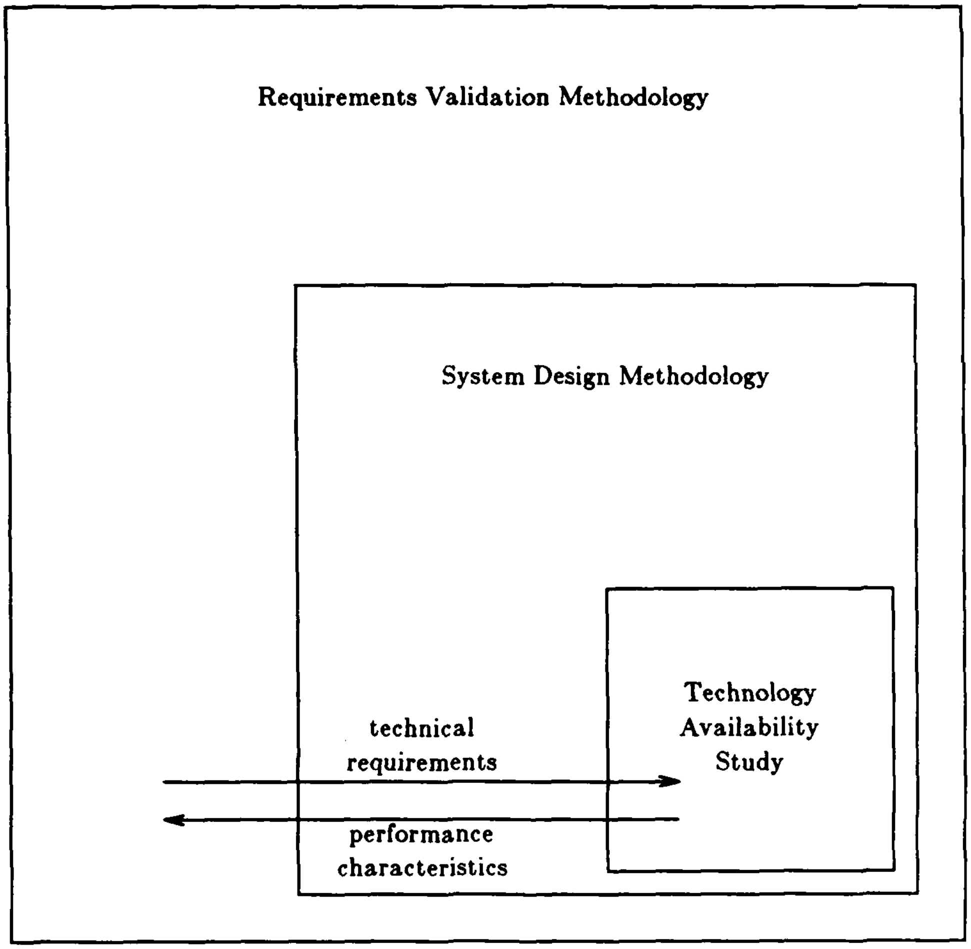

Requirements Validation Methodology
System Design Methodology
Technology Availability Study
technical requirements
performance characteristics

Figure 4.3-1 Technology Evaluation Paradigm

(1) Requirements Validation—assures that the production objectives are maintained as a foremost concern throughout the technology evaluation.

(2) System Design Methodology—assures systematic exploration of the number of design possibilities.

(3) Technology Availability Study—provides technical data needed to evaluate the different design alternatives.

A simplified view of the technology evaluation methodology allows us to describe the interdependency between its three methodology components in terms of two flows: a downward information flow depicting the propagation of technical requirements from the requirements validation to the technology availability study and an upward flow of information depicting the performance capabilities of various technologies, components, or the system as a whole.

Requirements validation contributes to a thorough understanding of the production requirements. The system design stage attempts to satisfy the needs identified through the requirements validation methodology. However, a class of designs created to meet production needs is also dependent on available technology. Therefore, the development of need supportive designs occurs in parallel with the development of an understanding of the type of technology that will be appropriate for the support of those designs.

For example, suppose that one needs to be able to review images quickly in some region of interest. This need translates into a requirement for fast retrieval over a local database. Then, in design simulation, it is discovered that what is really needed is a large primary memory rather than large disks. This realization directs what technologies might be of interest. Now it is necessary to perform a technology availability study to to find out if memories of the desired capacity and speed will become available. This study of technology availability has nothing to do with application, but the production needs have been propagated down as technical requirements through system design to the technology study that is necessary to determine what will satisfy production needs.

The bottom up flow is an upward propagation of performance characteristics from the technology availability study to the requirements validation. The results of the technology availability study include an understanding of what and when certain technologies of interest will be available; system design methodology will exploit these results for the purpose of developing a suitable class of designs. Each design is evaluated for performance, then information from these tests is fed into the requirements validation methodology. This way it can be determined what the consequences will be on the production system of a certain design with a particular technology.

The two flows just described form a feedback loop which cycles until the requirements validation is satisfied. Needs are identified for each methodology component and propagated downward. As studies are completed, the performance characteristics determined through each methodology propagate upward. As current requirements are satisfied it becomes more apparent which production requirements are necessary and which are unimportant. This, in turn, generates new requirements. By shortcutting the need/solution/new-need cycle our approach brings about significant development savings and increases the confidence that a delivered system exploiting some particular technology will satisfy the anticipated production needs.

There are several features that lend strength to our approach.

(1) The methodology we propose allows traceability between requirements, design, and technology. This is important in order to know the reasons behind the decision to select a particular technology and discard others. It is fundamental to the efficiency of the feedback loop to know why a particular decision was taken at all times, because to go through the decision process again in reevaluation is a needless expense.

(2) Requirement relevance is maintained throughout the technology evaluation. The design and technology studies are driven by production requirements and all technical results are fed back into the requirements validation.

(3) As stated earlier, a solution identifies new needs; so the requirements change as information is gathered. Requirement reinterpretation and redefinition due to technology impact is part of the process.

(4) The system design methodology is formulated to allow exploration of a large domain of possibilities and forces explicit decisions concerning why certain design types are accepted and why others are not. Implicit in the decision process is the relation of design to technology, or ways of using technology. So in effect, while exploring the field of design opportunities, we are also exploring the field of technology.

(5) More accurate results are expected if the technology evaluation is focused on specific production type and design class. The guidelines for the technology evaluation studies are established by the requirements and design studies. This way the study is ultimately driven by considerations regarding the needs of production.

(6) By separation of concerns, different groups of experts are able to integrate their knowledge. This not only facilitates evaluation, but effectively utilizes human resources and skills.

4.3.3. Methodology Description

This section describes our methodology for technology evaluation. The section begins with an overview of Requirements Validation, System Design, and Technology Availability Study. Following the overview is a detailed discussion of the System Design and Technology Study. Requirements Validation is discussed in detail in Section 3.

The purpose of the methodology is to establish a relationship between technology and the production capability of a proposed system, in the context of a particular organization. By production capability, we mean the amount and type of services the system will provide, the length of time it will take to provide the services, and the impact of the system on other operations within the organization. The relationship between technology and production capability is necessarily time dependent, because technology is constantly evolving. The organization in which the system will be used must be considered, because it determines the environment in which the system will operate.

In general, the production capability of a system depends on three things: *Human Factors*, or the needs of users who will interact directly with the system (such as operators and data entry personnel); the anticipated *workload*, or quantity of information the system will be expected to process and store, and any constraints imposed on processing time; and the *performance* of the system, which is the length of time the system will require to process a fixed quantity of information.

The methodology has three components: *Requirements Validation* relates human factors, workload, system performance, and production impact. *System Design* investigates system organizations appropriate for given requirements and technologies and provides system level performance information to Requirements Validation. Finally, *Technology Study* investigates available technologies and provides performance and other information at the component level for use in the System Design. As discussed in Section 4.3.2, we can view the three components of the methodology as forming a three-level hierarchy. System Design and Technology Evaluation respectively are contained within Requirements Validation and System Design.

It is important to note that the Technology Evaluation methodology differs from many TSD methodologies in two ways. First, its purpose is to relate requirements, types of technology, production impact, and *classes* of appropriate designs, rather than to produce a single system design. In general, we wish to identify the largest collection of designs which satisfy requirements and exploit available technology. Second, the flow of information from Requirements Validation through System Design to Technology Study is two-way. An initial statement of requirements is necessary to establish the scope of the Technology Study, but the requirements can often be expected to change based on information obtained from System Design and Technology Study.

4.3.3.1. Methodology overview

The Technology Evaluation Methodology has three major components: Requirements Validation, System Design, and Technology Study. In this section we present a brief overview of each component. Following the overview are sections which discuss in detail the System Design and Technology Study components, and discuss the need for automated support of the methodology.

The purpose of Requirements Validation is to describe the interaction between Human Factors, system workload, and performance, and to relate them to the production impact of the finished system. This is best accomplished by simulating system behavior and using information from the simulation to determine production impact. Aspects of the system which should be simulated are functionality, the human interface, and system-level performance. Correctness of the human interface can be evaluated by involving in the simulation the users who will closely interact with the system. Performance information obtained from the System Design component must be considered in order to realistically simulate response time. System workload is provided by the initial requirements analysis. It is a function of the environment, and is usually not subject to change during Technology Evaluation. The exception is when the Technology Evaluation shows that no available technology is capable of building a system which will meet the workload, subject to given performance constraints.

The purpose of System Design is to find system organizations appropriate for a given application and given classes of technology. It is this stage which builds the link between processing and functional requirements. In most cases, the use of various technologies will make possible a number of different system architectures. For example, the Technology Study may identify a new array processor, several times faster than existing array processors, which will be available one year in the future. It may be that the system would satisfy its performance requirements using a single fast processor, but would require several processors at the current level of technology. In this case, the choice of a particular technology would have a significant effect on system design.

System Design relates various technologies and system organizations which are appropriate to the problem at hand. Detailed component-level performance, cost, and availability information is obtained from the Technology Study. System Design identifies feasible architectures and integrates component-level information into information about the system as a whole. System level performance and availability information are then used to assess the production impact of the finished system. Both functional and performance characteristics are used by the simulations in the Requirements Validation stage to assess production impact and human factors. While performance is important to human factors considerations, it should be noted that not all connections between design and human factors involve only performance characteristics. For example, the use of a particular input device may be possible only with one architecture.

The Technology Study serves three purposes. First, it provides general performance and reliability information about various technologies. Second, it details the same information about individual components. Third, it provides information about the evolution of technology. Technological evolution involves such factors as when new technology will be available and changes in the cost of existing technology. This information is used both to determine the feasibility and cost effectiveness of using a particular technology, as well as how the use of a technology will affect delivery of the finished system. The delivery date in turn influences the system's production impact. This concludes the overview of the Technology Evaluation Methodology. The next section discusses in detail the System Design and Technology Study components. Following that is a discussion of automated support for the methodology.

4.3.3.2. System Design and Technology Evaluation

This section details the System Design and Technology Study components of our Technology Evaluation Methodology. The major part of the discussion will deal with System Design. Technology Study, which investigates characteristics of specific technologies and individual components, is tied closely to the particular technology being investigated, and does not lend itself to formalization. We consider Technology Study to be part of the Inference step of System Design, discussed later in this section.

System Design is a stage of the Total System Design (TSD) framework, discussed in Section 4.2. In the TSD view, System Design is broken into two phases: The *System Architecture Design* phase is responsible for producing overall system organization. without selecting any specific products to be used in building the system. Given detailed requirements established previously, System Architecture Design breaks the system into a number of functional hardware and software components, and connections among components. It is in this phase that we resolve such issues as hardware/software tradeoffs and the distribution of data and functionality in distributed systems. In general, the approach to System Architecture Design is the same in Technology Evaluation as in TSD, but the goals are different. In the traditional TSD view, the System Architecture Design phase is responsible for producing a single architecture which can be realized in subsequent phases. In Technology Evaluation, the goal is to explore as fully as possible the domain of possible solutions. This usually means producing a number of architectures, each of which exploits available technologies in a different way.

The second phase, *System Binding,* is responsible (in TSD) for selecting specific hardware and software products to satisfy the component requirements established in the System Architecture Design phase. Off-the-shelf components are procured and detailed requirements are established for components which must be customized or designed. In Technology Evaluation, we are more concerned with relating technologies to system organizations than with selecting a mix of specific products. Although it may occasionally be necessary to commit to using a specific product, we feel that the System Binding phase is not critical to Technology Evaluation.

A general description of the TSD Framework appears in Section 4.2. In the rest of this section we describe the System Architecture Design phase of the System Design Stage as it applies to Technology Evaluation. The Technology Study is considered part of the System Architecture Design phase, and is discussed below under the Inference step. The TSD Framework identifies several groups of activities, called steps, which are essential to any phase of development. They were first formulated in the context of system development, but they are equally important to Technology Evaluation. We will present the System Design component of the methodology in terms of these steps. In some places, consecutive steps are combined for sake of clarity.

In the first two steps, FORMALISM SELECTION and FORMALISM VALIDATION, we select a conceptual model which allows us to reason about system design, and a notation system for expressing design results. The model and notation system are chosen in the Formalism Selection step and their effectiveness for the problem at hand is evaluated during Formalism Validation. We have found that most design models currently in use are inadequate in that they do not fully cover the relationship between hardware, application software, and the operating system. This relationship is important because the interplay between these three elements affects system performance. In some application areas, such as business data processing, the effect of hardware and operating software on performance can be ignored. However, Geographic Data Processing includes many performance-critical applications (such as image processing, feature extraction, and accessing large data bases) and all factors which affect performance must be considered.

As a solution to the inadequacy of current modelling techniques, we propose the VIRTUAL SYSTEM model, described in detail in Section 4.4. The Virtual System supports the need for careful reasoning about application software, operating system, and hardware by modelling systems as a collection of interacting processes at three conceptual levels. The top, or FUNCTIONALITY level models application software, and views the system as it will be seen by application programmers. The lowest level, the ARCHITECTURE, models the physical machines on which the software runs. Between Functionality and Architecture is the SCHEDULER level, which is responsible for allocating resources and controlling execution of software at the Functionality level. The Scheduler corresponds roughly to the operating system in conventional computer systems, but is more general. Its purpose is to form a connection between Functionality and Architecture, and thus may include some functions which are implemented outside of traditional operating system software. For example, Scheduler functions may be implemented in language interpreters and communication packages. In general, the Scheduler provides a run-time environment for application software at the Functionality level. Although it is possible to view firmware as a separate system component, we do not include it in the model because, at the level of abstraction we are considering, its functions can be adequately modelled by Scheduler and Architecture.

Performance specification is included as an integral part of the Virtual System Model. Its role is to encapsulate any assumptions the designer makes about the performance and behavior of system components and any performance parameter definitions.

Having discussed what an appropriate conceptual model might be, we now address the EXPLORATION step, in which one searches for system organizations appropriate to the problem at hand. It is this step which is responsible for conducting the broadest possible search of the solution space. In deriving and classisfying potential designs, it is important not to discard any promising solutions. In order to avoid repeating work, records of discarded designs should be maintained, along with the reasons they were discarded. The various designs should be classified according to system structure and performance characteristics, and the design classes should be related to various technologies. It is essential to build such taxonomies of potential designs because, in Technology Evaluation, the domain of possible solutions is so large that the study becomes unmanageable if designs are not organized according to important system characteristics. As in the TSD Framework, the nature of the Exploration step is essentially creative and does not lend itself to formalization or automation.

The ELABORATION, CONSISTENCY CHECKING, and VERIFICATION steps are responsible for giving form to and ensure correctness of the results of Exploration. The Elaboration step records results of Exploration using some appropriate notation. Consistency Checking ensures semantic correctness of results expressed in the notation and detects improper use of the model. The Verification step checks proposed designs against the functional requirements they are expected to satisfy. Inadequate designs are discarded or returned to the Exploration step for modification. Both analytic techniques, such as formal verification, and experimental techniques, such as simulation. are used to ensure that ali requirements are met. Verification deals only with functional requirements; performance characteristics are evaluated separately.

The activities performed in these three steps are sufficiently complex as to require automated support. The number of possible inconsistencies which occur in even a small design is large enough to make manual checking of designs unfeasible. Similarly, the requirements which must be met by a realistic design, and the interplay between requirements and system organization, are too complex to be confidently validated without some amount of automated support. The next section provides a detailed discussion of the type of support we believe to be necessary for Technology Validation.

In the EVALUATION and INFERENCE steps we determine the system-level performance characteristics of various designs, determine what technology is required to realize each design, and assess the impact that each design will have on individual aspects of the human interface. The Evaluation step integrates information about the performance of specific components into performance information for the system as a whole. This information is then used in the Requirements Validation stage to assess the production impact of various designs. As in the TSD Framework, the purpose of the Inference step is to assess the impact of work done in previous steps. In the context of Technology Evaluation, this is accomplished by determining the types of technology required to realize various designs, as well as the cost and availability of each technology. The Technology Study, which determines the feasibility of using various technologies, is part of the Inference step. Because the goal of Technology Evaluation is to determine the usefulness of various technologies, there is a closer coupling between the Evaluation and Inference steps here than in traditional TSD view of system design.

The INTEGRATION step is responsible for combining the results of subordinate Technology Evaluations. It is often desirable to invoke a separate study for certain subsystems or types of technologies. For example, the Exploration step could partition a system into a number of large subsystems, and separate design teams could concurrently work with each subsystem. It is also possible to do separate design studies for different classes of design, and then build relationships between them. When subordinate evaluations are complete, this step is invoked to combine the results of each into information about the system as a whole. The Integration step in Technology Evaluation is consistent with the same step in TSD, in which the entire methodology can be invoked for subsystems which are regarded as "black boxes" early in the design process.

This concludes our discussion of the System Design Stage of the Technology Evaluation Methodology. The following section discusses the need of automated support for the activities described here.

4.3.3.3. Technology Evaluation support

This section discusses the need for automated support of the Technology Evaluation Methodology. The complexity of the Technology Evaluation process, especially in Requirements Validation, and the Consistency Checking, Verification, and Evaluation steps of the System Design component make automated tools essential for proper use of the methodology. Most of the discussion in this section will deal with automated support for the System Design stage. The Technology Study is highly dependent on the type of technology being investigated. It is difficult to formulate a general standard technique for this component, and we will not consider automated support for it in this section.

It is generally accepted that the tool kits available to today's designers are extremely inadequate. These are the most significant shortcomings: (1) most conceptual models of distributed systems do not fully represent the interplay between application programs, operating software, and hardware; (2) simulation languages currently in use allow for a large conceptual distance between system design specifications and system models; (3) even a small change in the design requires a large amount of work to make the system model consistent; and (4) the semantics of many modelling languages are not precisely defined, which makes it difficult to reason about designs.

If a system design environment is to be useful, it must use a model which closely reflects the physical reality of the system being designed. The semantics of the modelling language must be defined with sufficient precision to allow reasoning about the design. Finally, it must provide automated support to manage the complexity of the design effort. This indicates the need for a new type of system design environment, one rooted in the Virtual System Model discussed earlier. The remainder of this section discusses several important issues that must be addressed in the development of such a design environment.

FORMAL SEMANTICS: Well-defined semantics are essential to any modelling language. In the absence of formal semantics, precise reasoning about designs expressed in the language is difficult, and automated analysis is impossible. Formal semantics are also necessary to ensure that a system model, once built, is consistent with respect to certain modelling criteria. These principles, formally defined, become the requirements for tools used to check the consistency of system designs.

VERIFICATION: Verification deals with using formal methods, or informal methods rooted in formal models, to prove properties of system designs. Because it recognizes the interaction between software, executive, and hardware, the Virtual System Model provides a convenient framework for proving properties of software in conjunction with the rest of the system. A problem with existing methods of verification is the lack of ability to deal with the complexity of large systems. This implies the need for an approach that we call *proof management,* which involves breaking system verification into a number of pieces which are small enough to be manageable. Specifically, this involves:

- Using proofs at high levels of abstraction to guide proofs at more concrete levels. Applying verification techniques throughout the process of stepwise refinement should be easier than waiting until the entire design is complete.

- Using proofs about the software in isolation to guide proofs about the system as a whole.

- Reusing proofs about library components.

- Using verification techniques to devise strategies for testing designs.

SIMULATION: Most systems are too large to be verified solely by analytic means. This implies that simulation is necessary for complete confidence in a design. Two types of simulation are necessary:

- *functional simulation,* which is concerned with functionality, without regard to performance, and

- *discrete event simulation,* which provides performance information.

Both types of simulation may be generated directly from the Virtual System, and changes in the design are reflected immediately in the simulation.

USER INTERFACE: The designer must be provided with simulation results in the most easily understandable form. Textual information is usually not sufficient to describe system characteristics, especially those dealing with performance. Also, the functional simulation of the user interface should be as complete as possible. This often implies the need of graphic input/output devices which will mimic those used in the finished system.

INQUIRY/ANALYSIS CAPABILITIES: The support system should include tools for analyzing designs and simulation results. It should keep track of various designs and simulation results and allow the designer to correlate design changes with changes in performance and functionality. Simulations should be interactive and allow the designer to express queries and to change parameters while the simulation is running. The support system should also be able to evaluate hypotheses about the design from its definition and simulations.

ABILITY TO REUSE PREVIOUS WORK: Building system models is a costly, time-consuming task. The cost of designing a new system would be significantly reduced were it possible to use work already done in a previous design. The Virtual System Model is organized so that both components and knowledge about them (e.g. proofs, tests) can be reused. The support system should permit models of system components, along with important characteristics (functional properties and performance), to be stored in libraries. Ideally, a designer building a new system would begin by examining the library for any models which could be used in the new design. Models from the library would be combined with the larger system. They could be specialized (such as models of storage allocation and communication) or more general (such as models of large subsystems). The library could also store models of varying degrees of complexity. For example, a simple model of storage allocation might report only the amount of storage allocated, while a more complex one would keep track of the size and location of each page in virtual memory and the cost of swapping.

4.4. SPECIFYING SOFTWARE/HARDWARE INTERACTIONS

4.4.1. Introduction

We view a distributed system as a collection of application software modules, allocated over hardware components. This allocation, generally the responsibility of some operating system, may be static or dynamic: in a dynamic allocation, the mapping between software and hardware changes with time; in a static allocation, this mapping is constant. In this view, the system behavior is determined primarily by the software. Parts of the system other than software can affect its performance, however. For example, faults in the hardware may limit system capabilities, or the interaction of software and hardware in a particular allocation may degrade system performance. The workload, which is determined by the environment with which the system interacts, also affects system behavior.

Our work is aimed at developing system level models that enable the designer to formulate and answer questions regarding the system's logical correctness and performance characteristics when the interaction between the hardware and the software is important, i.e., when the impact of faults, failures, communication delay, hardware selection, scheduling policies, etc., must be considered. In the simplest terms, our concern extends beyond the traditional software correctness questions by addressing the issue of

> *employing logical verification techniques to determine software correctness and performance characteristics when running on a particular distributed hardware architecture and using a particular operating system.*

The models are called *Virtual Systems* [1] and represent either abstractions of existing systems or definitions of proposed systems. A virtual system consists of six components, each abstracting some aspect of a distributed system. The *Functionality* is an abstraction of the processes which carry out the system function (e.g. the applications software). The *Architecture* captures the overall hardware organization and distribution of the system. The *Scheduler* together with the *Allocation* define the relationship between functionality and architecture, and the changes which the relationship undergoes with time. This concept is an abstraction that encompasses many of the responsibilities one generally associates with an operating system, e.g., static and dynamic allocation of functions (in the functionality) to processors (in the architecture) as well as the allocation of time and space on an individual processor to the functions associated with it. The *Performance Specification* is an abstraction of both measurement probes and workload characteristics (the environment model is an integral part of the virtual system). The performance specification may be used to explicitly state the assumptions made by designers regarding the characteristics of the environment and of the system components to be utilized in the realization of the

[1] This work extends results published with M. S. Day under the title "Multifaceted Distributed Systems Specification Using Processes and Event Synchronization," *Proc. of 7'th Int'l Conf. on Soft. Eng.*, pp 44-55, 1984

system. The performance specification is coordinated with the rest of the model via the *Instrumentation.*

We are currently experimenting with specifying virtual systems using a language called CSPS *(Communicating Sequential Processes with Synchronization).* An extension of Hoare's CSP [1], CSPS allows synchronization between multiple processes in addition to the I/O primitives of CSP. The functionality, architecture, scheduler and performance specification are defined as closed communities of *communicating concurrent processes* while the allocation and instrumentation are defined as sets of *event synchronization rules.* The allocation, for instance, captures the interaction between software modules, hardware components and scheduling policies by specifying synchronization rules between events in the functionality, scheduler and architecture. In this manner the effect of the architecture and scheduler on the software execution is factored into the proofs. It should be noted, however, that we view CSPS only as an experimental notation scheme that allows one to exercise the modelling concepts prior to the development of a distributed system design language. Employing CSP as a base allows modelled systems to be verified using techniques already developed for verifying CSP programs [2,3,4] and leads to the emergence of a uniform incremental strategy for verifying both logical and performance properties of distributed systems. Consequently, work on performance evaluation, resource allocation, and verification of concurrent processes may be drawn together by reducing some problems from the first two areas to equivalent problems in the third.

The remainder of the report starts with a review of CSP which is immediately followed by an overview of the CSPS notation. Next, the structure of the virtual system and the motivation for that structure are explained. A simple ongoing example is used to illustrate the different components of a virtual system. The example is followed by an outline of the incremental proof strategy used to verify conformance to various types of functional (e.g., logical correctness) and nonfunctional (e.g., fault tolerance) system requirements. A discussion of the history of this effort and the pragmatics involved in modelling realistic systems concludes the report.

4.4.2. CSP Review

A full description of CSP is available in Hoare's original paper [1]. This section provides an informal definition of the syntax and semantics of the language. Three issues are addressed here: how to define a sequential process, how to specify concurrent execution of two or more processes, and how to describe communication between concurrent processes.

Each process definition takes the form

P :: S

where P is the process name and S stands for a composite statement. Ignoring communication, S is built out of three kinds of basic statements: assignment, guarded selection and guarded iteration. The assignment operator is ":=", while guarded selection and iteration assume the following syntax, respectively,

[b1 → S1 # b2 → S2 # b3 → S3]

*[b1 → S1 # b2 → S2 # b3 → S3]

where b*i* is a boolean expression called a *guard* and S*i* is an arbitrary sequence of statements.

A guard is said to be *passable* if the boolean expression evaluates to *true* and *fails* otherwise. A guarded selection fails if all the guards fail. If one or more guards are passable, the nondeterministic selection of one of the passable guards is followed by the execution of the corresponding guarded statement. A guarded iteration is exited if all the guards fail. If one or more guards are passable, the nondeterministic selection of one of the passable guards is followed by the execution of the corresponding guarded statement and by another iteration.

Concurrent execution of a finite set of processes is specified using the statement

[P1 ∥ P2 ∥ ... ∥ Pn]

where no process may reference any variables subject to change in any other process.

Communication is accomplished via two I/O commands: *send* (e.g., P!x -- meaning *send the value of x to process P*) and *receive* (e.g., Q?x -- meaning *receive from process Q a value to be assigned to x*). I/O transfers take place only when one process names another as its destination for output, and that process names the first as its source for input. Such a situation is called a *matching* pair of I/O commands. If one process is ready to send to or receive from a process which is not yet ready to match the issued I/O command, the former must wait until both processes are ready.

One of the most important features of CSP is that in addition to the boolean expression a guard may also include an I/O command as in the example below:

[b1; P!x → S1 # b2; Q?y → S2 # b3 → S3]

When an I/O command is present, a guard is passable if the boolean expression is *true* and a matching I/O command is pending; and fails if the boolean expression evaluates to *false* or the process named in the I/O command is terminated. This last rule is referred to as the *distributed termination convention.*

4.4.3. CSPS Concepts and Notation

CSPS *(Communicating Sequential Processes with Synchronization),* as used in this report, is a direct extension of CSP. The new feature, n-way synchronization, is motivated not by the desire to make a contribution to language theory but by practical considerations rooted in the modelling approach we have been pursuing. Every decision regarding notation is aimed at providing easy specification and modification of CSPS models. This section shows how n-way synchronization is specified in CSPS.

4.4.3.1. Labelled n-way synchronization

N-way event synchronization is defined as the simultaneous occurrence of N events. An *event* consists of an atomic action such as the execution of a simple statement, the selection of a guard, or the execution of an I/O command. The selection of a guard that includes an I/O command together with the execution of the respective I/O command are seen as a single event.

In CSPS event synchronization is indicated through the use of *synchronization commands*. The simplest form of a synchronization command consists of a *synchronization label* and a *synchronization operator,* (e.g., $)

```
P1:: ... a $; x:=0; ... a $ P!x; ...
P2:: ... *[ x<6; a $ → x:=x+y ] ... a $ b $; ...
P3:: ... *[ x<6; a $ Q?y → x:=x+y ] ...
```

where *a* and *b* are two synchronization labels.

A synchronization command may occur many times in the text of a single process. Any process whose text includes a particular synchronization command must always participate in the corresponding synchronization. A synchronization takes place only when each participating process is ready to execute the same synchronization command. If two or more synchronization and I/O commands appear together separated by blanks into a *composite synchronization command,* all the commands must occur together—this feature enhances modularity of the models but adds no extra power to CSPS. CSPS extends the distributed termination convention to cover event synchronization commands. The last element in a guard may be an I/O command or a synchronization command or a composite synchronization command. A guard fails if the boolean part is *false,* a terminated process is involved in any of the synchronizations, or I/O commands appearing on the guard; a guard is passable if the boolean part is *true,* the I/O command may be executed and all synchronizations may take place.

One might argue that two-way synchronization is easily described using I/O commands that pass dummy data and that probably n-way synchronization could be simulated by two way synchronization and, therefore, adds unnecessary complexity to the language implementation. Although we have been able to show that n-way synchronization may be simulated by employing I/O commands, it is our contention that even when two-way synchronization suffices the use of synchronization commands offers the advantage of a cleaner separation of concerns, ease of understanding and greater flexibility. Furthermore, for describing virtual systems, n-way synchronization is more intuitive and provides much needed economy of expression.

To illustrate the advantages of the notation we are proposing, let us consider a situation that might occur when modelling a real-time system consisting of some control software that drives a motor. At some point in the system design one needs to model both the software and the motor. Because the design may undergo several iterations and because, once developed, the motor model should be available for use in future designs, it is desirable to have it independent of any particular software with which it might be interfaced. A specification that satisfies this requirements may look as follows

```
M::     [ on:=false; off:=true
             * [    on → run                          #
                    on → on:=false; off:=true         #
                    off → idle                        #
                    off → on:=true; off:=false        ]      ]
```

It provides a nondeterministic behavior description of the motor which must be interfaced with the control software logic whose task could be, for instance, to turn on and off the motor. Given the notation introduced earlier, the interfacing may be accomplished by simply adding two synchronization labels on the guards controlling the transitions between the running and idling states. (The distinction between the original model and the additions is clearly visible and reversible!)

```
M::     [ on:=false; off:=true
             * [    on               → run                          #
                    on; STOP $       → on:=false; off:=true         #
                    off              → idle                         #
                    off; START $     → on:=true; off:=false         ]      ]
```

If at some later date it becomes necessary to control the rate (as discrete rotations let's say) a label STEP could be added on the first guard (on; STEP $ → run). The same label could be used as feed-back rather than control, i.e., to allow the software to count the number of rotations the motor executes, while the labels START and STOP may be used also to control a status light:

```
SL::    [ light:=false
             *[     START $      → light:=true        #
                    STOP  $      → light:=false]      ]
```

The main motivation for n-way synchronization, however, rests with the need to match, i.e., synchronize, instantiations of the same event at three levels in the system: application software, operating system and hardware. This will be illustrated fully in later sections of the report.

4.4.3.2. Unlabelled n-way synchronization

Our search for notational convenience also led to the introduction of the "$$" as a synchronization command interpreted as: "*a process must resynchronize with every process with which it synchronized since the last unlabelled synchronization or since the start of the process if no unlabelled synchronization has taken place yet.*" More than one synchronization operator may be used if selective resynchronization is desired, e.g., "$$" would require resynchronization with processes synchronized via a "$" but would ignore any occurrence of a synchronization bearing another operator symbol such as "@".

The motivation for this type of synchronization rests with the fact that often what appears at the software level to be an atomic action may involve an entire sequence of events at the hardware level. The unlabelled synchronization provides a convenient way of acknowledging the end of the sequence of hardware level events without having to introduce additional labels:

software level action: STOP $; turn_motor_off; $$;

motor model actions: on; STOP $ → on:=false; off:=true; $$;

By convention the software level action above may be also written in the following equivalent compact form

STOP $$ turn_motor_off;

4.4.3.3. N-Way synchronization with pattern match

N-way synchronization, as introduced so far, is unable to simulate the data transfers accomplished by I/O commands. To simulate

```
P::    ... Q!x ...
Q::    ... P?y ...
```

where x and y are integers, would require an infinity of labels, each encoding one possible value being transferred. This is an unacceptable limitation since the interdependencies between the software and the hardware may not be limited to control information but must consider the data being processed. The *n-way synchronization with pattern match* is a mechanism for accomplishing what may be seen as data transfer.

Here is an example of how to simulate an I/O exchange using synchronization with pattern match where x and y are assumed to be boolean:

Version 1 (special case, feasible when the domain of x and y is finite):

```
P::    ...    PtoQ $ (x); ...
Q::    ... [  PtoQ $ (true)  → y:=true    #
              PtoQ $ (false) → y:=false   ] ...
```

Version 2 (general solution for simulating I/O exchanges):

```
P::    ...    PtoQ $ (x); ...
Q::    ...    PtoQ $ (y'); ...
```

PtoQ is the synchronization label which is used to determine (a priori) the set of processes that must synchronize. The pattern definition appears as a list following the synchronization operator. For each variable in the pattern absence of a quote indicates that the *value* of the particular variable is part of the pattern for the respective synchronization; a single quote indicates that the value of the particular variable is part of the pattern for the respective synchronization but it is *indeterminate,* i.e., the variable will assume any value (within the restrictions of the variable type) that renders the pattern match successful. If a value is selected and the synchronization occurs the variable is actually assigned the particular value and the value may be used in the execution of following statements. The assignment of a value to an indeterminate variable does not affect the truth value of the boolean part of a guard.

A synchronization with pattern match is successful if a synchronization could occur in the absence of any patterns and the patterns associated with each instance of the label match. Two patterns match if there is at least one assignment of values to the indeterminate variables which would result in making the list of values identical. Given, for instance,

```
(1)    K $ (x,y',z);   where x=2, y=4 and z=4

(2)    K $ (a',b,c);   where a=8, b=7 and c=4

(3)    K $ (u',v,w');  where u=3, v=9 and w=1
```

(1) and (2) match leaving a=2 and y=7 but (1), (2) and (3) fail to match because of the inability to reconciliate b=7 with v=9. (When several synchronizations with pattern match are combined, the pattern evaluation proceeds from left to right. Since this feature is not used in the report we will not discuss it any further.)

To illustrate a typical use of synchronization with pattern match we will show one way to provide the motor status to the control software:

```
M::    [ on:=false; off:=true
           * [    STATUS $ (on)      → skip                    #
                  on                 → run                     #
                  on; STOP $         → on:=false; off:=true         #
                  off                → idle                    #
                  off; START $       → on:=true; off:=false         ]      ]
```

Although the synchronization with pattern match makes the use of I/O commands obsolete, we chose to retain both mechanisms in CSPS. CSP is used to describe models of various hardware, application software and operating system components with the I/O commands specifying communication between components present in the same layer, e.g., application software. Interactions between the application software and the operating system and between the operating system and the hardware are described using synchronization commands. By maintaining this distinction the specification of a virtual system is easier to understand, configure (utilizing preexisting models of various components), modify and analyze.

4.4.4. Virtual System

A *virtual system* captures the functionality, architecture, scheduling policies and performance attributes of the system it models. This is accomplished by structuring the virtual system in terms of four communities of communicating processes that are related to each other via event synchronization. Communication within each community takes place via CSP I/O commands. The components of a virtual system are as follows:

> *Functionality* is a model of the application software and consists of a community of communicating processes called *functions*.

Architecture is a model of the hardware organization and consists of a community of processes called *processors*.

Scheduler is a model of the run time environment supporting the application software (i.e., operating system, language interpreter, etc.) and consists of a community of processes called *schedules*.

Allocation is a model of the behavior interdependency between the application software, hardware and operating system making up a complete, potentially distributed, system. The allocation consists of *event synchronization rules* between the functionality, architecture and scheduler. Each rule is identified by a unique synchronization label and is specified by the placement of synchronization commands bearing that label in the three communities of processes.

Performance specification is a formal definition of measures and assumptions (including behavior patterns) used by the designer in the analysis of a virtual system and consists of a community of (mostly noncommunicating) processes called *actors*.

Instrumentation is a formal definition of the manner in which measures and assumptions are attached to the components of the virtual system. The instrumentation consists of synchronization rules between events in the functionality, architecture and scheduler, on one hand, and events in the performance specification, on the other.

The next four sections expand on the motivation for the structural components of the virtual system and provide simple illustrations based on a producer/consumer problem.

4.4.4.1. Functionality

The *functionality* represents the requirements for the application software. As such, it defines the interactions between the system and its operating environment as observed at the system user level. (This particular work makes no distinction between system functions and operating environment functions, but such a distinction could be made.) The software requirements are given by means of an operational model whose structural properties are relevant only when considered in relation to a particular architecture and scheduler, as shown two sections later.

Although the designer chooses to break down the functionality in one particular way in order to take best advantage of the available or postulated processors, there are many aspects of the functionality that may be evaluated by considering the functionality alone, e.g., logical correctness with respect to some externally stated criteria, freedom from behavior anomalies such as deadlock, and some primitive performance characteristics (e.g., relative frequency of execution of certain functions for some class of environmental inputs). Although these proofs assume no interference from the operating system and the hardware and the availability of adequate resources (e.g., storage space), they may be used later in proofs, about the system as a whole.

To start the example let us consider a producer P which generates several values and passes them, one at a time, to a link L; the link L forwards each value to a consumer C; the consumer C receives a value from link L and computes the sum of all the values received so far. This functionality may be described as follows:

```
[ P || L || C ]

where

P::    [ x:=0  *[ x<3 → x:=x+1; L!x ]      ]

L::    *[ P?y → C!y ]

C::    [ u:=0  *[ L?z → u:=u+z ]    ]
```

The specification is written in CSP, is self-contained and makes no reference to any operating system and hardware capabilities or resources.

4.4.4.2. Architecture

The *architecture* models the physical structure of the system and the behavior of the individual components that make it up. The components are not seen as physical devices but as resources required to support the application software. Associated with each model there is a *viewpoint* defining the level of abstraction and the purpose of the model. Different types of components are recognized at different levels of abstraction (e.g., nodes in a network, individual machines, large hardware components, etc.) and alternate models are used for each component depending upon the issue one choses to address (the dynamics of storage management, communication behavior, fault detection, etc.).

The example for this section is communication via failing lines between two sites. An originator O selects a value to be sent to destination D, and sends it on one of the unreliable interconnections I1, I2:

```
[ O || I1 || I2 || D ]

O::     [ n:=0; up:=true
               *[ up →        n:=n+1; up:=false
                              *[      not(up); I1!n    → up:=true    #
                                      not(up); I2!n    → up:=true    ]      ]      ]

I1::    [ up1:=true    *[     up1; O?k1       → D!k1                #
                              up1             → up1:=false          ]      ]

I2::    [ up2:=true    *[     up2; O?k2       → D!k2                #
                              up2             → up2:=false          ]      ]

D::     [ m:=0         *[     I1?r            → m:=m+1     #
                              I2?r            → m:=m+1     ]      ]
```

The message being transmitted is modelled by a message sequence number. The unreliable nature of the interconnections has been expressed as a nondeterministic choice in a guarded iteration. Selection of the first guard results in transmission of a value from O to D. Selection of the second guard results in the termination of the process modelling the respective interconnection.

In writing a processor model one must exercise great care to ensure the validity of the model. What may appear to be equivalent behaviors in one context may prove to be different in another. By using termination to model the interconnection failure, for instance, the model states that this particular *failure may be detected* by the other processors via the distributed termination convention, e.g., O could find out about a failure in I1 by using the statement

```
*[ I1!n → skip ]
```

which exits whenever I1 is terminated. An undetectable error could be modelled by entering an infinite loop

```
I1::    [ up1:=true    *[     up1; O?k1       → D!k1                #
                              true            → up1:=false          ]      ]
```

Other methods are blocking the process by issuing an I/O command which will never be matched by the named process and failing the process by the use of an *abort* statement.

The model of the originator O illustrates another subtle implication of our architecture. The status of the interconnection is checked prior to selecting it. If this were not so, the model of the originator might take the form:

```
O::     [ n:=0; up:=true
               *[ up →        n:=n+1; up:=false
                              *[      not(up) → I1!n; up:=true      #
                                      not(up) → I2!n; up:=true      ]      ]      ]
```

The change is that the I/O commands are guarded by *true*, rather than being part of the guards. This corresponds to selecting an interconnection for transmission without testing it to determine if it is functioning, and can result in deadlock: if the first guard is selected and I1 has failed, I1 will never ask for input from O and the originator will wait forever.

4.4.4.3. Scheduler and allocation

The *scheduler* and the *allocation* describe the interdependency between functionality and architecture. The scheduler consists of a community of processes called schedules. Each schedule, together with the event synchronization rules that are part of the allocation, establishes a mapping between relevant events in one of the functions in the functionality and sequences of events involving one or more processors in the architecture.

To illustrate the event mapping idea let us consider an APL program as a realization of functionality and an APL machine (hardware interpreter) as a realization of architecture. Both functionality and architecture have states: for the APL program, the state depends on the value of local data and and the line number of the statement being interpreted; for the interpreter, the state depends on the program being interpreted, the interpreter's local data, and the program counter of the APL machine. In this common situation, the states of the program are mapped onto the states of the machine so that for every state of the program there is a corresponding distinct state of the machine. There may be unmapped intervening states in the machine, but not in the program. Each state transition in the program requires a sequence of state transitions in the machine until the states match again. The only differences between this illustration and the mapping defined by schedules are the presence of unmapped states in the functionality and the disregard for any events not involved in a n-way synchronizations. Given for instance the following event sequences

```
f1    f2    f3    f4    f5    in the function F
|                       |
s1    s2    s3    s4    s5    in the schedule S
      |     |     |     |
q1    q2    q3    q4    q5    in the processor Q
```

where the vertical bar indicates a synchronization, event f1 in the function F is mapped to the sequence <q2,q3,q4> of events in the processor Q.

The allocation is said to be static whenever for each function F there is a unique schedule which always maps events in F to sequences of events in some unique processor P. The allocation is said to be dynamic whenever the events in F may be mapped into sequences of events involving more than one processor.

The nature of the scheduler depends upon the viewpoint adopted for the architecture and upon the characteristics of the function to processor mapping. This section illustrates three representative instances of this mapping: static one-to-one function/processor mapping, static many-to-one function/processor mapping and dynamic one-to-one function/processor mapping.

4.4.4.3.1. Static allocation (one-to-one)

The static allocation of functions to processors does not require meaningful schedules if the processor models are very simple. The static allocation of the producer-consumer functionality to the origin-destination architecture with link L being allocated to interconnection I1 may be described by synchronizing directly events in the functionality with events in the architecture.

```
[ P || L || C || O || I1 || D ]

P::     [ x:=0  *[ x<3 → x:=x+1; po $; L!x  ]        ]

L::     *[ li $ P?y → C!y ]

C::     [ u:=0  *[ L?z → cd $; u:=u+z           ]        ]

O::     [ n:=0; up:=true
                 *[ up; po $ → n:=n+1; up:=false
                               *[      not(up); I1!n   → up:=true   #
                                       not(up); I2!n   → up:=true   ]       ]       ]

I1::    [ up1:=true    *[      up1; li $ O?k1 → D!k1                #
                               up1              → up1:=false        ]       ]

D::     [ m:=0         *[      I1?r     → cd $; m:=m+1      #
                               I2?r     → cd $; m:=m+1      ]       ]
```

The label po (may be read P on O) matches the production of a new x in P to the incrementing of the message counter n in O while the label cd matches the consumption of a new z in C to the incrementing of m in D. Similarly, L is allocated to I1 using the label li which matches a data passage through L to a data passage through I1. Because I2 is not used, it has been left out of the model and, in order not to have to change the descriptions for O and D, any reference to I2 is treated as a reference to a terminated process. Leaving I2 as an active component of the model could have caused undesirable behavior (erratic transmissions between D and O via I2).

4.4.4.3.2. Static allocation (many-to-one)

The sharing of a single processor by several functions may be illustrated by introducing two producers in the model used earlier.

```
[ P1 ‖ P2 ‖ L ‖ C ‖ O ‖ I1 ‖ D ‖ S ]

P1::    [ x:=0  *[ x<3 → x:=x+1; po1 $; L!x ]          ]

P2::    [ x:=0  *[ x<2 → x:=x+1; po2 $; L!x ]          ]

L::     *[       li $ P1?y → C!y        #
                 li $ P2?y → C!y        ]

C::     [ u:=0  *[ L?z → cd $; u:=u+z          ]       ]

O::     [ n:=0; up:=true
                *[ up; po $ →  n:=n+1; up:=false
                               *[      not(up); I1!n   → up:=true    #
                                       not(up); I2!n   → up:=true    ]      ]      ]

I1::    [ up1:=true     *[     up1; li $ O?k1  → D!k1              #
                               up1             → up1:=false        ]      ]

D::     [ m:=0          *[     I1?r     → cd $; m:=m+1      #
                               I2?r     → cd $; m:=m+1      ]      ]

S::     *[       po1 $ po $ → skip      #
                 po2 $ po $ → skip      ]
```

The schedule S maps both P1 and P2 on processor O but imposes no particular policy with regard to the use of this shared resource. S could be modified, however, to permit P1 and P2 to take turns.

```
S::     [ live1:=true; live2:=true
                *[ live1 or live2 →
                        live1:=false
                        *[ not(live1); po1 $ po $ → live1:=true]
                        live2:=false
                        *[ not(live2); po2 $ po $ → live2:=true]       ]      ]
```

The complexity of the schedule is due to the need to avoid having one producer blocked by the termination of the other.

4.4.4.3.3. Dynamic allocation

The redefinition of the function/processor mapping may be embodied in the schedule's logic and is usually triggered by some local condition that develops in some part of the system. Our illustration for this section involves the static mapping of P and C to O and D, respectively, and the dynamic allocation of L to the interconnections I1 and I2. L is mapped to I1 until the failure of I1 causes L to be mapped to I2. This version of the example includes also the use of synchronization with pattern match and unlabelled synchronization.

The synchronization with pattern match is employed with the label po in order to make O transmit to D the same value that P sends to C in place of the message number used earlier. This is accomplished by having O use an indeterminate variable assignment (po $ (n');) which induces n in O to assume the same value as x in P (po $ (x);), i.e., the only value for which a match becomes possible. In the case of the label cd the pattern serves only as a check that the value z received by C is the same as the value r received by D.

The unlabelled synchronization is used to block L from accepting any more data until I1 (or I2) has completed the data transfer from O to D and involves either L, I1 and S or L, I2 and S, depending which interconnection is in use.

```
[ P || L || C || O || I1 || I2 || D || S ]

P::    [ x:=0  *[ x<3 → x:=x+1; po $ (x); L!x        ]        ]

L::    *[ li $ P?y → C!y; $$ ]

C::    [ u:=0  *[ L?z → cd $ (z); u:=u+z        ]        ]

O::    [ up:=true
              *[ up; po $ (n') →
                              up:=false
                              *[      not(up); I1!n.   → up:=true   #
                                      not(up); I2!n    → up:=true   ]      ]      ]

I1::   [ up1:=true    *[      up1; li1 $ O?k1          → D!k1; $$    #
                              up1                      → up1:=false  ]      ]

I2::   [ up2:=true    *[      up2; li2 $ O?k2          → D!k2; $$    #
                              up2                      → up2:=false  ]      ]

D::    *[      I1?r    → cd $ (r)    #
               I2?r    → cd $ (r)    ]

S::    [       *[ li $ li1 $ → $$      ]
               *[ li $ li2 $ → $$      ]       ]
```

This particular version of the example will be used later to illustrate the incremental proof strategy. For this reason, it is important to point out that when one ignores the n-way synchronizations, for the purpose of analyzing a community of processes outside the context of the particular virtual system, any occurrence of an indeterminate variable (e.g., n') is treated as a random assignment of some value of the proper type.

4.4.4.4. Performance specification and instrumentation

The *performance specification* consists of a community of processes called actors whose interactions with the other components of a virtual system, defined by the *instrumentation*, take two distinct forms:

(1) they embody assumptions made by the designer about the characteristics of existing or envisioned system components—this is accomplished by stating the rules by which various relevant performance attributes are being computed;

(2) they control nondeterminism in the functionality, architecture, and scheduler by inducing patterns of behavior having particular characteristics.

When actors are used in a measurement capacity, they play a similar role to the reporting components of simulation languages. Events in the functionality, architecture, scheduler and even performance specification may trigger predefined actions in the information recording actors. Take, for instance, the computation of the flows through I1 and I2. An actor A may take advantage of the synchronization labels li1 and li2 to determine which interconnection is used and to store this information into some internal counters f1 and f2.

```
A::     [ f1:=0; f2:=0
              *[      li1 $ → f1:=f1+1      #
                      li2 $ → f2:=f2+1      ]       ]
```

If the flows are available, other data such as the average delay associated with the transmission via L may be deduced.

Fundamental to obtaining correct results is the need to ensure that a recording actor does not actually interfere with the behavior of the processes with which it is synchronized. Proving this for A is trivial: A is always ready to synchronize on li1 and li2 without imposing any restrictions in their ordering. B is provided below as a counterexample.

```
B::     [ f1:=0; f2:=0
              *[ true →
                      li1 $; f1:=f1+1;
                      li2 $; f2:=f2+1 ]       ]
```

This actor forces alternate use of I1 and I2—the result is deadlock after the first use of the interconnection I1 by the link L. A sufficient condition for noninterference is to assure that the synchronized statements in a recording actor can be executed in arbitrary order, e.g., A permits the behavior {<li1>, <li2>}* while B is limited to the behavior {<li1,li2>}*. (Note: <a1,a2,a3> denotes a sequence and {<a>,<b>,<c>} denotes a set of sequences.)

Nondeterminism may enter the model because of modelling a nondeterministic activity in the system or its environment (e.g., arrival of system requests) or a deterministic process whose original deterministic nature has been lost in the process of abstraction into the model. In many cases, something is known about the statistical nature of the nondeterminism present. Actors may be defined so as to have a probabilistic behavior which, through instrumentation, may be imposed on functions, processors and schedules.

To illustrate the way in which actors may impose a particular behavior pattern on other components of the system, let us consider an actor W whose role is to induce the failures of I1 and I2 after a fixed number of transmissions on each. (A probabilistic version of W could be defined but, for the sake of brevity, we chose to present a deterministic version of W.)

```
I1::   [ up1:=true   *[   up1; li1 $ O?k1     → D!k1        #
                          up1; fi1 $          → up1:=false  ]       ]

I2::   [ up2:=true   *[   up2; li2 $ O?k2     → D!k2        #
                          up2; fi2 $          → up2:=false  ]       ]


W::    [ f1:=1; f2:=1
              *[    f1<3; li1 $ → f1:=f1+1       #
                    f1=3; fi1 $ → skip           #
                    f2<5; li2 $ → f2:=f2+1       #
                    f2=5; fi2 $ → skip           ]       ]
```

The new labels fi1 and fi2 are used to force I1 and I2 to terminate.

Many of the features available today in discrete event simulation languages such as SIMULA [5] (e.g., random distributions for event arrival and processing time, reporting features, etc.) could be easily introduced in CSPS. This suggests that event synchronization should be given serious consideration as a potential mechanism for integrating discrete event simulation in design specification languages. The analogy to actual instrumentation of system components has a certain intuitive appeal. The ability to integrate the performance and functional issues while still maintaining a strong separation of concerns is desirable and has been attempted already by others (e.g., SREM [6]).

4.4.5. Verification Strategy

Our ultimate goal is to establish the basis for a system design specification language which offers the designer not only expressive power but also a variety of convenient logical verification techniques able to address both correctness and performance considerations in a uniform manner. Because of the size and complexity of real systems these techniques must be amenable to the development of incremental proofs and to recycling parts of existing proofs. Moreover, top-down system design makes it desirable to relate proofs corresponding to different levels of abstraction.

Our contributions toward achieving these ambitious goals have been limited so far to adapting CSP verification methods [2,4] for use with CSPS and to outlining a general strategy for the development of incremental proofs of system properties. The remainder of this section illustrates our proof strategy on a slightly modified version of the example given under dynamic allocation. For the sake of brevity, the proofs are presented in an outline form. There are two parts to the illustration. The first one is concerned with the correctness of the software specification in isolation. Because n-way synchronization is not involved in this proof, it serves also as a review for the CSP verification method adapted for use in this section. The proof of the correctness of the system as a whole follows.

Among existing techniques, we found Soudararajan's use of *communication traces* (called communication sequences in [4]) both easy to use and compatible with the overall objectives of our work. Each process has an associated trace whose purpose is to record, in order, all the I/O commands executed by the process (including the exchanged values) and the termination event *end*. When considering only I/O commands, a trace consists of pairs ((e),(v)) where e is a label uniquely identifying the pair of processes involved in the I/O and the direction of the data transmission (e.g., P sending to L may be indicated by PL) and v is the value being transmitted. (Note: the definition given in [4] has been modified to accommodate synchronization commands.) A partial correctness proof starts by proving appropriate properties about the individual processes in isolation; next, a rule of parallel composition is used to prove properties of the community.

All the axioms and rules of inference are fully described in [4] and will not be repeated here. Instead, we provide a proof outline for the functionality consisting of the processes P, L and C. We start by supplying pre- and post-assertions for each process.

```
        { Tp=<> }
P::     [ x:=0  *[ x<3 → x:=x+1; L!x        ]        ]
        { Tp=<((PL),(1)),((PL),(2)),((PL),(3)),((end),(nil))> }

        { Tl=<> }
L::     *[ P?y → C!y ]
        { Value(Tl[PL])=Value(Tl[LC]) }

        { Tc=<> }
C::     [ u:=0  *[ L?z → u:=u+z       ]        ]
        { u=SUM_OVER(Value(Tc[LC])) }
```

where

(1) Tp, Tl, and Tc are the traces for P, L and C, respectively;

(2) <> denotes the empty sequence;

(3) *nil* is a place holder indicating the absence of a value;

(4) Tl[PL], called the projection of Tl with respect to PL, denotes a trace obtained from Tl by eliminating every pair whose first element does not contain PL and by replacing the remaining pairs with pairs containing only PL as first element and the value matched to PL as the second element;

(5) Value(Tl) denotes a sequence obtained from Tl by replacing every pair by its second element; Type(Tl) works the same way but it keeps the first member of the pair; Size(Tl) may be used to get the length of the trace;

(6) SUM_OVER(seq) is a function that totals all the values appearing in some sequence of integers.

Applying now the parallel composition rule, one may deduce

{ true }
[P ‖ L ‖ C]
{ Tp=<((PL),(1)),((PL),(2)),((PL),(3)),((end),(nil))> and
Value(Tl[PL])=Value(Tl[LC]) and
u=SUM_OVER(Value(Tc[LC])) and
Compatibility(Tp,Tl,Tc) }

But in this case

Compatibility(Tp,Tl,Tc) => Tp[PL]=Tl[PL] and Tl[LC]=Tc[LC]

leading to

{ true } [P ‖ L ‖ C] { u=6 }

One more thing remains to be proven: termination. In our example this is rather easy to show: P terminates after three passes through the guarded iteration (due to strictly monotonic increase of the value of x); P's termination cascades to L and C (due to the *distributed termination convention).*

Next we need to consider the impact of the scheduler and the architecture—one would like to prove that the selected architecture and scheduling policies do not change the functionality of the system. The approach is the same. Properties of processors and schedules are proven in isolation and then the parallel composition rule is applied to all the processes making up the virtual system. Synchronization commands are treated in the same manner as the I/O commands except that (1) the value part of the trace is defined as the matched pattern if a synchronization with pattern match is involved and as *nil* otherwise, and (2) if several synchronization and I/O commands are involved, the pair entered in the trace contains as a first element a list of all the labels and involved and as a second element a list of corresponding values, e.g., ((li1,OI1),(nil,1)).

An outline of the proofs for the individual processes in the virtual system is given below. Please note, that for the sake of simplicity the unlabelled synchronizations have been eliminated. This triggered some code replication in L, the elimination of the label li, and the introduction of the labels di1 and di2. Otherwise, the system as a whole is unchanged.

```
        { Tp=<> }
P::     [ x:=0  *[ x<3 → x:=x+1; po $ (x); L!x        ]         ]
        { Tp=<((po),(1)),((PL),(1)),((po),(2)),((PL),(2)),((po),(3)),((PL),(3)),((end),(nil))> }

        { Tl=<> }
L::     *[      li1 $ P?y → C!y; di1 $#
                li2 $ P?y → C!y; di2 $]
        { Value(Tl[PL])=Value(Tl[LC]) }

        { Tc=<> }
C::     [ u:=0  *[ L?z → cd $ (z); u:=u+z      ]         ]
        { u=SUM_OVER(Value(Tc[LC])) and Value(Tc[LC])=Value(Tc[cd]) }

        { To=<> }
O::     [ up:=true
                *[ up; po $ (n') →
                                up:=false
                                *[      not(up); I1!n   → up:=true   #
                                        not(up); I2!n   → up:=true   ]         ]         ]
        { Value(To[po])=Value(To[OI1;OI2]) }

        { Ti1=<> }
I1::    [ up1:=true     *[      up1; li1 $ O?k1         → D!k1; di1 $ #
                                up1                     → up1:=false ]         ]
        { Value(Ti1[OI1])=Value(Ti1[I1D]) }

        { Ti2=<> }
I2::    [ up2:=true     *[      up2; li2 $ O?k2         → D!k2; di2 $ #
                                up2                     → up2:=false ]         ]
        { Value(Ti2[OI2])=Value(Ti2[I2D]) }

        { Td=<> }
D::     *[      I1?r    → cd $ (r)      #
                I2?r    → cd $ (r)      ]
        { Value(Td[I1D;I2D])=Value(Td[cd]) }

        { Ts=<> }
S::     [       *[ li1 $ → di1 $        ]
                *[ li2 $ → di2 $        ]         ]
        { true }
```

where

(1) To[OI1;OI2] denotes a trace obtained from To by eliminating every pair whose first element does not contain OI1 or OI2 and by replacing the remaining pairs with pairs containing only OI1 and OI2 (alone or together) in the first element and the values matched to OI1 and OI2 as the second element.

The application of the parallel composition rule generates

```
{ true }
[ P ‖ L ‖ C ‖ O ‖ I1 ‖ I2 ‖ D ‖ S ]
{ Tp=<((po),(1)),((PL),(1)),((po),(2)),((PL),(2)),((po),(3)),((PL),(3)),((end),(nil))> and
  Value(Tl[PL])=Value(Tl[LC]) and
  u=SUM_OVER(Value(Tc[LC])) and Value(Tc[LC])=Value(Tc[cd]) and
  Value(To[po])=Value(To[OI1;OI2]) and
  Value(Ti1[OI1])=Value(Ti1[I1D]) and
  Value(Ti2[OI2])=Value(Ti2[I2D]) and
  Value(Td[I1D;I2D])=Value(Td[cd]) and
  Compatibility(Tp,Tl,Tc,To,Ti1,Ti2,Td) }
```

Given the structure of this virtual system one may immediately write

```
Compatibility(Tp,Tl,Tc,To,Ti1,Ti2,Td)
      =>    Tp[PL]=Tl[PL] and Tl[LC]=Tc[LC] and
            To[OI1]=Ti1[OI1] and Ti1[I1D]=Td[I1D] and
            To[OI2]=Ti2[OI2] and Ti2[I2D]=Td[I2D] and
            Tp[po]=To[po] and Tc[cd]=Td[cd] and
            Tl[li1]=Ti1[li1]=Ts[li1] and Tl[di1]=Ti1[di1]=Ts[di1] and
            Tl[li2]=Ti2[li2]=Ts[li2] and Tl[di2]=Ti2[di2]=Ts[di2]
```

which states that every pair of matching I/O traces must be identical and that every pair or triplet of matching synchronization traces must be identical.

Several important conclusions may be drawn from this post-condition:

(1) u=6—which is what software correctness required.

(2) Value(To[OI1;OI2])=Value(Td[I1D;I2D])—which states that the sequence of values received by D must be the same as the one sent by O, regardless of which interconnection was used. The label cd ensures that if this condition is not met the system blocks and therefore it never terminates.

(3) Size(To[OI1;OI2])=Size(Td[I1D;I2D])=3—which states that three transmissions occurred via I1 and I2 before they terminated.

None of the conclusions we have drawn so far hold if the virtual system does not terminate—we did a partial correctness proof and we have not shown total correctness. Any attempt to prove termination, however, runs into difficulty due to the nondeterministic behavior of I1 and I2 which may terminate prematurely and thus block L. Under these circumstances it is important to reformulate the entire issue of termination by asking the questions:

> *What conditions must be met in order to assure termination?*
> *What condition prevents termination and where?*

These are the types of questions a designer must answer every time a new design proposal is put forth; successive refinements will eventually lead to a virtual system for which termination may indeed be proven.

We found all the techniques in use today to be of limited value in answering this kind of questions—they follow a paradigm where one must first determine all the circumstances under which blocking may be present and later prove that none of the situations can actually occur. Since the first step is where the designer is most likely to fail when analyzing a complex system, design errors may remain undetected. It appears to us, however, that the behavior information contained in the communication traces could be exploited to a greater extent in order to deal with some aspects of termination.

Considering our example again, it is rather easy to observe that the communication behavior of all the processes may be abstracted as regular sets, if one ignores the values being transmitted:

$$
\begin{aligned}
&\mathrm{Type(Tp)} && \subseteq && \{\langle \mathrm{po,PL,po,PL,po,PL} \rangle\} \times \{\langle \mathrm{end} \rangle\} \\
&\mathrm{Type(Tl)} && \subseteq && \{\langle (\mathrm{li1,PL}),\mathrm{LC,di1} \rangle, \langle (\mathrm{li2,PL}),\mathrm{LC,di2} \rangle\}^{*} \times \{\langle \mathrm{end} \rangle\} \\
&\mathrm{Type(Tc)} && \subseteq && \{\langle \mathrm{LC,cd} \rangle\}^{*} \times \{\langle \mathrm{end} \rangle\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{Type(To)} && \subseteq && \{\langle \mathrm{po,OI1} \rangle, \langle \mathrm{po,OI2} \rangle\}^{*} \times \{\langle \mathrm{end} \rangle\}. \\
&\mathrm{Type(Ti1)} && \subseteq && \{\langle (\mathrm{li1,OI1}),\mathrm{I1D,di1} \rangle\}^{*} \times \{\langle \mathrm{end} \rangle\} \\
&\mathrm{Type(Ti2)} && \subseteq && \{\langle (\mathrm{li2,OI2}),\mathrm{I2D,di2} \rangle\}^{*} \times \{\langle \mathrm{end} \rangle\} \\
&\mathrm{Type(Td)} && \subseteq && \{\langle \mathrm{I1D,cd} \rangle, \langle \mathrm{I2D,cd} \rangle\}^{*} \times \{\langle \mathrm{end} \rangle\}
\end{aligned}
$$

$$
\mathrm{Type(Ts)} \quad \subseteq \quad \{\langle \mathrm{li1,di1} \rangle\}^{*} \times \{\langle \mathrm{li2,di2} \rangle\}^{*} \times \{\langle \mathrm{end} \rangle\}
$$

By using the compatibility constraint one may develop the following necessary termination condition:

$$
\begin{aligned}
&\mathrm{Type(Tp)} && \subseteq && \{\langle \mathrm{po,PL,po,PL,po,PL} \rangle\} \times \{\langle \mathrm{end} \rangle\} \\
&\mathrm{Type(Tl)} && \subseteq && \{\langle (\mathrm{li1,PL}),\mathrm{LC,di1} \rangle\}^{*\mathrm{n1}} \times \{\langle (\mathrm{li2,PL}),\mathrm{LC,di2} \rangle\}^{*\mathrm{n2}} \times \{\langle \mathrm{end} \rangle\} \\
&\mathrm{Type(Tc)} && \subseteq && \{\langle \mathrm{LC,cd} \rangle\}^{*3} \times \{\langle \mathrm{end} \rangle\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{Type(To)} && \subseteq && \{\langle \mathrm{po,OI1} \rangle\}^{*\mathrm{n1}} \times \{\langle \mathrm{po,OI2} \rangle\}^{*\mathrm{n2}} \times \{\langle \mathrm{end} \rangle\} \\
&\mathrm{Type(Ti1)} && \subseteq && \{\langle (\mathrm{li1,OI1}),\mathrm{I1D,di1} \rangle\}^{*\mathrm{n1}} \times \{\langle \mathrm{end} \rangle\} \\
&\mathrm{Type(Ti2)} && \subseteq && \{\langle (\mathrm{li2,OI2}),\mathrm{I2D,di2} \rangle\}^{*\mathrm{n2}} \times \{\langle \mathrm{end} \rangle\} \\
&\mathrm{Type(Td)} && \subseteq && \{\langle \mathrm{I1D,cd} \rangle\}^{*\mathrm{n1}} \times \{\langle \mathrm{I2D,cd} \rangle\}^{*\mathrm{n2}} \times \{\langle \mathrm{end} \rangle\}
\end{aligned}
$$

$$
\mathrm{Type(Ts)} \quad \subseteq \quad \{\langle \mathrm{li1,di1} \rangle\}^{*\mathrm{n1}} \times \{\langle \mathrm{li2,di2} \rangle\}^{*\mathrm{n2}} \times \{\langle \mathrm{end} \rangle\}
$$

where $\mathrm{n1} \geq 0$, $\mathrm{n2} \geq 0$ and $\mathrm{n1}+\mathrm{n2}=3$.

From this it becomes apparent that premature termination of I1 and I2 will cause problems.

There are two reasons why this condition is not necessary and sufficient:

(1) Local behavior patterns within a single process may prevent termination. This could happen in I1, for instance, if one uses *true* in place of up1 on the second guard.

```
I1::    [ up1:=true    *[    up1; li1 $ O?k1     → D!k1; di1 $ #
                              true                → up1:=false ]
```

(2) The analysis ignored the pattern matches. With a different schedule, it is possible for D to receive values from O in the wrong order which would cause blocking between P and D on the synchronization cd. A more refined behavior analysis may be considered in such cases. In our example, by using the behavior of S we can show that the combined behavior of I1 and I2 (ignoring the end labels) may be described by

{<(li1,OI1),I1D,di1>}*n1 x {<(li2,OI2),I2D,di2>}*n2

from which it is easy to see that no blocking takes place at the synchronization cd.

The proof outline presented in this section is indicative of the potential that exists today to analyze distributed system designs by employing program verification methods and of the type of techniques that seem to be best suited to the task faced by the designer. The difficulty stems from the inherent complexity of the systems we try to describe, apparent even in the simple example used throughout the report. Significant reductions in the complexity of the proofs may be brought about by breaking the proof into small parts that require a limited context to be built and understood. The strategy we are pursuing involves three tracks:

(1) breaking the proofs into local and global issues on the line followed by Apt, et al., [2] and by Soundararajan [4] but considering two levels of locality, the process level and the virtual system component level, i.e., functionality, architecture, etc.;

(2) breaking proofs into levels of abstraction where a proof at one level is used to guide the lower level analysis as in the termination proof;

(3) proof reusability as illustrated by the functionality proof appearing as a part of the system proof.

4.4.6. Concluding Remarks

The motivation for the work presented in this report is rooted in the concept of a *Total System Design (TSD) Framework* [7]. The TSD Framework treats distributed systems as complex hardware/software aggregates whose design often requires careful consideration of the relationship between functionality and architecture in order to meet highly demanding constraints. The *virtual system* is a model that attempts to capture precisely the essence of this relationship.

The definition of the virtual system is motivated by traditional software engineering considerations:

(1) *Comprehensive coverage of key technical considerations.* Broad and integrated coverage of key technical considerations facing the distributed

system designer is provided, including software and hardware organization, static and dynamic allocation, and performance modelling. Furthermore, in contrast to other distributed system design methodologies [8,9] where the architecture is treated only as a collection of computational sites, the virtual system attributes to processors arbitrary computational characteristics and behavior.

(2) *Separation of concerns.* The components of the virtual system correspond to well-established areas of design expertise (e.g., software design, architecture design, resource allocation, performance modelling). The communities of communicating processes used to model the components capture design decisions that may be evaluated (up to a certain point) independently from the overall design or in a limited context. Dependencies between these components are expressed via synchronization rules that tie them together: e.g. functionality, architecture and scheduler (all having independent existence) are brought together by the allocation.

(3) *Complexity control.* The model permits independent elaboration, analysis and modification of its components while allowing easy identification of the entities affected by particular changes, when present. For instance, the architecture can be modified without changing the functionality, to determine how best to implement a required set of functions. Alternatively, the functionality can be restructured without changing the architecture to determine how best to take advantage of a particular architecture or feature of the system hardware. In either case, the definition of the allocation may be used to determine the consequences of that change for the other components.

(4) *Design flexibility.* There are systems whose software or hardware are given and others for which both must be selected by the designer. In the latter case, sometimes the architecture and other times the software becomes the dominant concern with its design turning into a constraint for the other. All these cases and combinations thereof are accommodated by the structure of the virtual system.

The first formal definition of the virtual system [10] was developed from a system theoretical perspective and lacked practicality, analyzability and parsimony. The use of processes and event synchronization has remedied the situation and brought us closer to what we envision a distributed system design language ought to provide. Most recent efforts have been directed primarily toward the development of a powerful, compact, unobtrusive and easy to modify notation and the definition of its semantics. This report illustrates some of the choices we have made but omits the more powerful aspects of the notation system which relay on an elaborate macro capability. The language work has been driven by a case study where a series of small but realistic virtual systems have been built. By comparison, the producer/consumer illustration is not representative of the details of the modelling process—in the example simplicity has been favored over realism.

In parallel with the CSPS experiments which are intended to help us build modelling expertise on realistic problems, we are also pursuing a number of related topics:

(1) Refinement of the CSPS language and of the incremental verification strategy.

(2) Development of linguistic constructs needed to model faults and to specify fault detection.

(3) Establishment of design decomposition paradigms for *dual stepwise refinement* of both software and hardware specifications and *hierarchical structuring* of virtual systems.

(4) Development of specialized hardware and operating system models that could be used to specify alternate system configurations with a minimum of effort on the part of the designer.

(5) Establishment of a disciplined approach and formal foundation for investigating hardware/software trade-offs in distributed systems.

Although our current work is centered around the use of CSPS as a specification language, we view CSPS only as a tentative notation useful in the exploration of a number of significant theoretical issues and leading to the emergence of a new type of design specification language, one that views the development of distributed systems from a *total system design* perspective.

4.4.7. References

[1] Hoare, C. A. R., "Communicating Sequential Processes," *CACM* 21, No. 8, pp. 666-677.

[2] Apt, K. R., Francez, N., and DeRoever, W. P., "A Proof System for Communicating Sequential Processes," *ACM Trans. Prog. Lang. & Sys.* 2, No. 3, pp. 359-385, 1980.

[3] Levin, G. M. and Gries, D., "A Proof Technique for Communicating Sequential Processes," *Acta Informatica* 15, No. 3, pp. 281-302, 1981.

[4] Soundararajan, N., "Axiomatic Semantics of Communicating Sequential Processes," *ACM Trans. on Prog. Lang. and Sys.* 6, No. 4, pp. 647-662, 1984.

[5] Birtwistle, G. M. et. al., *Simula Begin,* New York: Petrocelli, 1973.

[6] Bell, T. E., Bixler, D. C. and Dyer, M. E., "An Extendable Approach to Computer-Aided Software Requirements Engineering," *IEEE Trans. on Soft. Eng.* SE-3, No. 1, pp. 49-60, 1977.

[7] Roman, G.-C. et al. "A Total System Design Framework," *Computer* 17, No. 5, pp. 15-26, May 1984.

[8] Estrin, G., "A Methodology for Design of Digital Systems," *1978 NCC Proc.*, pp. 313-324, 1978.

[9] Mariani, P. M. and Palmer, D. F., *Distributed System Design*, IEEE Computer Society Press, 1979.

[10] Roman, G.-C. and Israel, R.K. "A Formal Treatment of Distributed Systems Design," *Requirements Engineering Environments*, Ohno, Y. (editor), pp. 3-12, OHM/North-Holland Pub. Co., 1982.

5. FAR-TERM MPE OPERATIONAL CONCEPT

This section proposes an operational concept for the Defense Mapping Agency's (DMA) far-term Modern Programming Environment (MPE). The MPE is a software development environment (SDE) which will be used to increase productivity at DMA.

5.1. INTRODUCTION

A *modern programming environment* is a coordinated collection of computers and software tools dedicated to supporting the development and maintenance of computer software. The goals of an MPE are two-fold: to increase the productivity of the software project personnel (analysts, designers, programmers, managers, etc.) and to improve the quality (reliability, correctness, maintainability, etc.) of the software that they develop and maintain. The MPE is "modern" in the sense that it seeks to integrate "state of the art" software and computer technologies into a coherent facility. Moreover, to remain modern, the MPE must be flexible enough to evolve gracefully as better technologies emerge. DMA contracted with the Data Systems Division of General Dynamics Corporation to develop the DMA MPE and introduce it into DMA operations. Our contract addresses how this initial facility should evolve.

The objective of Task 4 is to study the potential impacts of the changing MPE requirements in the context of currently identifiable technological trends. In this study we examine five technological areas:

- tools,
- human interfaces,
- methodologies,
- architectures,
- software development environments.

To make our study realistic, we concentrate on clearly identifiable trends and avoid purely speculative ideas on the future development of computer and software technologies.

In approaching this task, we assume a functionalist perspective. First, we identify the evolutionary processes to which the MPE facility will be subjected because of the changing needs of DMA. With this view of the changing requirements, we then derive an understanding of the structures and procedures that will most likely foster facility evolution. Using this understanding of the requirements and the technological trends, we then suggest an evolutionary path for the DMA MPE facility.

5.2. CHANGING REQUIREMENTS

In this section we discuss the directions we see product development moving and what role the MPE will fill in this process.

5.2.1. The Nature of the Production Process

The production process, as we see it, is divided into three main areas, the initial extraction and analysis of data, the storage of data and its later retrieval for further use, and the actual synthesis of data and product generation. As the products requested increase in sophistication, the production process will require changes in order to meet them, and the methods in each area will have to advance. The evolutionary path we see for each of these areas is reported in this section.

5.2.1.1. Data extraction and analysis

In this initial phase of the production process, features are extracted from the raw sensor data coming in, and stored away for later use. Recently, increased DoD needs have resulted in more types of data to be processed as well as significant increases in the amounts of data. In response to these requirements, the extraction process should evolve in the following manner.

(1) *Manual.* Currently, the dominant means for data extraction is a mostly manual process using hard copy of the data. This is a slow process which is rapidly becoming incapable of handling the increasing quantity of data, and therefore must be made more efficient to meet current and future needs.

(2) *Semi-automated.* By incorporating computer technology into the process, as is currently being attempted, the efficiency and hence capacity are increased. This is accomplished in several ways such as processing of the images in a variety of ways to make recognition of features easier by the operator and the automatic entry of the extracted data into the database through a suitable user-interface. Hence the productivity of the personnel is substantially raised to keep pace with increasing needs.

(3) *Automated.* As the volume of data continues to rise in the future, it will be necessary to decrease the amount of work personnel must do on each piece of incoming data. To accomplish this will require the computer systems to absorb more of the work previously done by personnel, especially in extraction of features from the data. These computer systems will necessarily be highly sophisticated, artificial intelligence systems capable of doing much of the reasoning required to extract the useful information from the raw data, and thereby minimizing the amount of operator interaction and increasing throughput.

5.2.1.2. Data storage and retrieval

Once the data has been extracted and initial analysis performed, it is necessary to save it in such a way that when it is needed later for a product, the data can be quickly and easily obtained. Again, as the amount of data increases, the sophistication of the storage and retrieval system must also increase in order to adequately serve its purpose, and we see it developing along the following lines.

(1) *Highly segmented, type/range checked.* The present mechanism is to make simple automatic checks on the features extracted from a single piece of data for consistency and then store them away in one of several different retrieval systems based on some criteria of the data.

(2) *Partially correlated, semantically checked.* The next step is a system which checks limited relationships between various extracted pieces of data for consistency and correctness, and saves these relationships in addition to the basic extracted data. This system requires much quicker access to the data in order to do these checks, effectively requiring much of the data to be directly accessible. By storing more than just the basic data, the system provides better assurance that the data is correct, and reduces the amount of time and work spent correlating data, since it need only be done once.

(3) *Integrated, knowledge inclusive.* In the far-term, all of the data will be integrated into a single, cohesive storage and retrieval system. In addition, knowledge about the data would be saved, and additional knowledge derived from existing knowledge and new data. This environment will provide much more support for advanced product generation to meet the future needs of DoD.

5.2.1.3. Data synthesis and product generation

Along with the proliferation of data to be processed comes the requirement to synthesize new types of information and products. As the number, variety, and sophistication of desired products increases, the product generation process must evolve to satisfy these needs. We view this evolution in the generation of products as taking the following path to meet its goals.

(1) *Single product.* At present, the dominant method is the generation of single products, whether standard or custom, to meet single needs. This method of product generation requires much effort per product, and hence limits the number of products that can be produced. Although for specific, special applications it may be necessary to build a custom product, in general the products will tend away from this area because of the cost.

(2) *Product families.* As development moves away from single products, the dominant trend is toward the development of a collection of similar of products from a central core product by the incorporation of a few custom parts to the core product. This generates a family of products from one

central piece of work, and the cost per product goes down, or conversely, more effort can be put into the core resulting in better products for the same cost.

(3) *Application knowledgeable.* In the future, highly specialized products would be produced from the generic core products by the inclusion of detailed application knowledge. This would allow for the development of many novel services and as yet unthought of products.

5.2.2. Required Technical Support

As the production process evolves, it is evident that technical support will also have to evolve in order to support the requirements of the production process. Each area mentioned in the previous section places its own set of requirements on the advancements in both software and hardware necessary to accomplish these goals, which are described below.

5.2.2.1. Data extraction and analysis

In this area, three types of systems are envisioned to facilitate each of the levels of computer support (manual, semi-automated and fully automated), namely bookkeeping, image processing, and artificial intelligence systems.

(1) *Bookkeeping systems.* The major type of computer support at the current time is a record keeping system. As each piece of data is extracted, it is characterized and put into the storage system, with a record being maintained as to where the data is, so that it can later be recovered for further use.

(2) *GDP/IP systems.* To facilitate the partial automation of the extraction process, image processing and display systems are introduced to support the personnel. These systems provide two major tools to the operator 1) real-time image manipulation and display capability to assist in the extraction of data, and 2) display interface for automatic entry of identified features, which substantially increase productivity. These systems must include high performance hardware to do the transformations in minimal time, high-quality displays to enhance image quality and ease recognition, and quality software tools to make the hardware readily usable by the operator.

(3) *GDP/AI systems.* To eventually have the computer system do the feature extraction with minimal operator intervention requires special purpose hardware and software capable of performing this function. This requires software that can do: deductive reasoning to determine which transformations need to be performed on particular images to extract the data, the actual extraction of the data, and other jobs the operator previously performed, and do them reliably. The hardware must be capable of running such software in addition to the image processing and display

software.

5.2.2.2. Data storage and retrieval

For the data storage systems, evolution will be from a system of simply storing syntactically correct pieces of data to one capable of reasoning about the data it contains. The support for this also evolves from a simple tape storage to one that integrates data and functionality.

(1) *Tape storage.* The current system consists of a large storehouse of magnetic tapes on which the various pieces of data are kept. When data is required, the correct tape is found and mounted on a computer system so the data becomes available. Although sufficient for a bookkeeping type of storage and retrieval system, its very slow retrieval times and inability to easily correlate pieces of data make it unsuitable for more advanced storage and retrieval systems.

(2) *Advanced storage data bank.* The increasing need to correlate and semantically check the stored data requires that all of the data be readily accessible. For this, movement to a new technology (such as bubble memory devices or optical disks) that has very large capacity and yet all of the data is available in a relatively short time to the processing system, is required.

(3) *Data/knowledge bases.* To eventually support a single, comprehensive data system, in which knowledge about the data is also maintained, requires that a sophisticated software system be built on top of an advanced hardware storage technology. The software system must be capable of maintaining the consistency and correctness of the data and their relationships, as well as doing reasoning about the data. These necessitate a data/knowledge base whose reasoning capabilities are geared toward interpreting geographic data.

5.2.2.3. Data synthesis and product generation

The wide variety of software needed in the product generation process requires the use of a software design methodology. Use of a methodology results in a more structured approach to software development, which in turn maximizes the effectiveness of software used for product generation.

(1) *Custom programs.* Programs for very specific purposes both old and new will still be used to generate products, but the trend will be away from these. Old programs will be retrofitted to merge better with new software, and the new software will be written to make retrofitting unnecessary in the future.

(2) *Reusable software.* In order to develop families of products, the underlying core software must be reusable. Most software today is not designed with this in mind, and a major modification to the software design process will

have to be made so that future software is designed for reusability. Reusability also decreases the amount of software that must be maintained, allowing for more time on new development.

(3) *Specialized high power systems.* As the sophistication of DoD needs rise, so the requirement to produce specialized systems will become stronger. Reusable software modules will be necessary to minimize the amount of new software that must be written to develop the specialized systems, in order to provide the quality, quantity and variety of systems desirable. Methodology development will be necessary to incorporate application details into the product development to create products for very specific needs.

5.2.3. The MPE Role

In this scene of changing requirements, the MPE has several very important roles, both in the maintenance of existing software and the development of new software. The maintenance comes mainly in two forms, production support software, and GDP systems software, while development of new software is primarily for new products.

(1) *Maintenance of production support software.* Most of the existing software is in FORTRAN and COBOL, and as time goes by it will be increasingly difficult to maintain these old programs. By bringing these older pieces of software under the umbrella of the MPE, the maintenance will be done in a consistent manner, providing for a longer and more useful life. In addition it provides for the integration of this older software with new software developments.

(2) *Maintainence of GDP systems software.* The MPE will maintain current and future GDP system software. This software will be in a variety of languages, such as C, Ada®, Lisp and Prolog, depending on the type of the GDP system (image processing, artificial intelligence, etc.).

(3) *New software development.* Development of new software under the MPE will facilitate its design for maintainability and reuseability. It will accomodate, in the future, the use of newer programming languages, such as Ada®, and it will allow current software written in FORTRAN and COBOL to coexist and cooperate with the new software in production support and products.

5.2.4. The MPE Factor

The MPE has three major effects on the production process: increase in quality and productivity, software reusability, and standardized development for maintainability.

(1) *Productivity and quality increases.* The MPE currently provides a consistent form for the software development cycle and a set of development tools to

assist the software designer. In addition to increasing the quality of individual software modules, the use of development tools simplifies the task of configuration management (CM), which is concerned with the interaction between modules. Through the use of these tools, mistakes and errors can be found early in the development cycle, when correcting them is much less costly and time consuming, resulting in higher productivity. Similarly, the MPE encourages fewer errors in the design process and aids in the correction of those made, leading to products of higher quality.

(2) *Software Reusability.* Continued use of the MPE will lead to new software that is reusable because the MPE methodology is designed with reusability of software as a major concern. This reduces development time for new systems by facilitating the use of existing software as much as possible, and also makes maintainability much easier since fewer distinct pieces of software need to be maintained.

(3) *Standardized new system development to assure easy maintainability.* A side effect of the MPE will be the imposition of standard system development methods on contractor developed systems. This is because the software will need to be maintained in a cost effective manner under the MPE.

5.3. TECHNOLOGICAL TRENDS

In this section we report the trends in computing technology that we see impacting the evolution of programming environments. As stated in a previous section, we examine five technological areas: tools, human interfaces, methodologies, architectures, and software development environments. For each area we present a taxonomy of the issues involved and define our terminology. We use these concepts in the recommendations stated in the next section.

5.3.1. Tools

A software tool is a program (or a set of related programs) which assists programmers with aspects of the development and maintenance of software systems. In this examination of the trends in software tools technologies, we use these classification criteria: capability, life-cycle integration, methodological support, applicability, objectives, and life-cycle phase support.

5.3.1.1. Capability

We can identify a promising trend toward increased capability in the software tools that are available. Part of this increased capability will come from improvements in the current techniques and part from application of new techniques. We see tools technology evolving from use of the familiar syntactic concepts to "newer" ideas that have not been widely applied to the tasks of developing and maintaining software. Accordingly, we classify the capabilities of tools into the following categories:

(1) *Syntactic processing.* Syntactic processing tools check and manipulate the structure of objects (programs, specifications, documents, etc.) based on a syntax for the object. For example, the SDDL (Software Design and Documentation Language) tool consists of a well-defined language and programs which process documents written in the language to produce several useful reports. This is a purely syntactic operation, but can be quite effective. We anticipate that syntactic processing concepts will continue to be important in the new tools that become available.

(2) *Semantic checking.* Tools which check the semantics (meaning) of objects and their interrelationships will be increasingly important in the future. For example, software tools which check the completeness and consistency of objects fall into this category.

(3) *Reasoning milieu.* Future tools need to go beyond syntactic and semantic checking and provide a milieu for reasoning about the properties of software objects. For example, such tools may assist a designer to verify the logical correctness of a design specification and determine whether the specification exhibits other useful logical properties. Our survey of this area indicates that, although much research is still needed to make such tools practical, these technologies hold great promise.

(4) *Expert assistance.* Artificial intelligence technologies, particularly expert systems, have received much attention in recent years. Although we remain skeptical of many of the claims of the AI proponents, we expect that expert systems concepts can be successfully applied in the design of software tools. In the long term, such experts can augment and amplify the work of the human software developers and maintainers.

(5) *Automatic generation.* The automatic generation of software from high-level specifications is an active topic of research and development today. Although currently available tools are not yet mature and efficient enough for production usage, they do show promise in many restricted application domains. Automatic programming tools are not a panacea for all that ails the software development community, but they can be an enormous aid to the human software developer.

5.3.1.2. Life-cycle integration

The software tools within a programming environment should be integrated across the software life-cycle. That is, the tools should be coordinated so that understanding of the software product flows smoothly from problem definition through maintenance and from requirements specifications to code. We see a promising trend toward increased integration of the tools across the life-cycle. For this study, we identify three levels of integration:

(1) *None.* Most of the currently existing tools are not integrated across life-cycle stages.

(2) *Traceability.* Traceability means the ability to cross-reference items in the system requirements specification with items in the design specification (and perhaps the actual code). This can be done today, but in the future we expect that this can be more fully automated.

(3) *Requirements/design integration.* The key to life-cycle integration of design activities rests with the ability to relate design and requirements specifications. Requirements work is not limited to the problem definition stage. The design of each component of a system entails the specification of both functional and non-functional requirements for a set of subcomponents from which the component will be eventually assembled. By including subcomponent requirements as a part of the design specification, designers can hierarchically partition the design verification process, i.e., the designer can show that a component satisfies its requirements whenever its subcomponents satisfy their requirements. Tools can be built which support this process.

5.3.1.3. Methodological support

Most tools are built around some view of the software development and maintenance process. The tools may be:

(1) *Bound to a methodology.* Many tools are designed to support a specific methodology. The tools assume the world view fostered by the methodology and are highly specialized toward the needs of users of the methodology.

(2) *Supportive of a class of methodologies.* Other tools may support a whole class of methodologies. The methodologies in a class may be based on similar concepts or utilize similar techniques. For example, a tool built around a dataflow view of system structure might be able to support any methodology that uses dataflow as a central concept.

(3) *Independent of any methodology.* Other tools may be independent of all methodologies. For example, database and text processing tools are useful in the software development and maintenance process regardless of the methodology used. General tools which can be customized to support specific needs of methodologies are especially useful. The SDDL package, for instance, allows users to define keywords that are specific to the methodology being used.

For the foreseeable future, we believe all three paths of tool development will'yield useful results.

5.3.1.4. Applicability

Historically, software engineering researchers and tool developers have paid greater attention to the needs of the developers of new software than to the needs of maintainers of existing software. Because software maintenance is really software redesign, these efforts have assisted in the maintenance process somewhat. However, tools are needed which address the unique needs of software maintainers. Recently more emphasis has been placed on maintenance.

5.3.1.5. Objective

Software tools have one of two possible objectives: synthesis and analysis of software systems.

(1) *Synthesis.* Synthesizing tools help software engineers to build software systems. Such traditional tools as editors, compilers, and linkers fall into this category. But more exotic tools such as rapid prototyping systems, automatic program generators, and design expert systems are also included in this category.

(2) *Analysis*. Analytical tools assist with the evaluation of software systems. They help designers to ascertain the properties of a system or its specification. For example, performance evaluation tools help software designers to analyze the performance characteristics of a system (either existing or proposed). Proof management tools can assist in the verification of logical properties (e.g, correctness, freedom from deadlock, etc.) of a system specification.

5.3.1.6. Life-cycle phase support

Often software tools are specialized to support a single phase of the system life-cycle, e.g., problem definition; other tools may support multiple phases. The early program development tools (e.g., assemblers, compilers, linkers, debuggers, etc.) were concentrated on the coding and low-level testing of programs. More recently tools have been developed to support the other phases—particularly the design phases.

5.3.2. Human Interfaces

Human/computer interfaces are the channels through which information is exchanged between a computer user and the computer. These interfaces are the subject of much research. Recent advances in hardware and software technology and in the understanding of human behavior and perception are opening new possibilities for human/computer interaction. Human interfaces are important to our study because the industrial success of any programming environment is heavily dependent on its attention to human factors. In this study, we investigated trends in the style and modes of interaction and in accessibility to information.

5.3.2.1. Style of interaction

The style of a human interface captures the manner and format in which information is presented to the user by the computer and vice versa. At present there are two primary styles of interaction with programming environments: the traditional linguistic style (one-dimensional) and the increasingly important graphical style (two- or three-dimensional).

(1) *Linguistic*. In the linguistic style, information is exchanged between the computing system and the human user in a textual format. This style is easy to implement on nearly any computer available today. The linguistic style can also be formalized. Designers of human interfaces can use the extensive theory of formal languages to precisely define the nature of the human/computer interaction. Also, natural (human) language interfaces are an area of extensive research and development within the AI community.

(2) *Graphic*. In the graphical style, information is conveyed via pictorial displays and positioning movements. Information is represented in a compact (but perhaps less exact) form that can be perceived quickly by

humans. The ready availability of low-cost, high-quality graphics hardware devices has made graphical and icon-based interfaces practical.

5.3.2.2. Mode of interaction

The mode of interaction describes the level of user involvement with a processing activity, e.g., the execution of a software tool. Early software tools are essentially "batch" processes; more recent tools allow more direct involvement by users. We divide modes of interaction into four categories:

(1) *Static.* In a static mode of interaction, users set up a processing activity (prepare the input for a tool), initiate it (run the tool), then examine the results when it is complete. Users are not involved during the processing.

(2) *Dynamic.* The dynamic interaction mode is similar to the static mode. However, the dynamic mode enables users to trace the internal processing actions as well as to examine the results of an activity.

(3) *Interactive.* An interactive mode enables users to guide a process by exchanging information with the process while it is active. The user actions and the processing actions are viewed as separate entities which interact with one another.

(4) *Synergistic.* Synergistic interfaces transcend the capabilities of interactive interfaces to merge the user's actions with those of the computer; the computer becomes (metaphorically) an extension of the user. The user and the computational activity work in the same syntactic and semantic environment, i.e., the structure and meaning of objects are the same to both. By merging the capabilities of tools with those of the human programmer, the effectiveness of each is magnified.

5.3.2.3. Accessibility to information

Software development environments are information systems. They organize, store, and process information about the objects created and modified during the software development and maintenance cycles. We can characterize human interfaces by how accessible this information is to the user. The human interface can provide three levels of accessibility:

(1) *Localized.* With localized access users have access only to information inside of their current contexts, e.g., to information contained in the file that they are currently processing. There is no formal organization or cross-referencing of related information from different contexts.

(2) *Structured.* Structured access organizes the different information environments in some well-defined way (perhaps hierarchically). Although users may not have direct access to related non-local information, they can

follow a well-defined path of context changes to access the information.

(3) *Total.* Total access means that all related information is immediately available to the user regardless of the context in which it is specified.

The evident trend is toward the more sophisticated types of information accessibility.

5.3.3. Methodology

Good methodologies are probably the most important components of any attempt to improve the productivity of programmers and the quality of their products. A software engineering methodology provides a systematic view of the software development and maintenance and establishes rules for structuring the process. In our examination of trends in this active area of research, we look at two classification criteria for methodologies: their flavor and their relation to tools.

5.3.3.1. Flavor

Methodologies can normally be characterized by one of the following orientations:

(1) *Specification oriented.* These output-oriented methodologies stress the development of documentation at each phase. The content and syntax of these documents are important aspects of specification-oriented methodologies.

(2) *Process oriented.* Process-oriented methodologies emphasize the steps to be followed in solving a problem. For example, a process-oriented methodology may emphasize a problem solving technique such as rapid prototyping or production rules.

(3) *Automatic generation.* Automatic generation methodologies seek to synthesize a program from high-level descriptions of the system. In such a methodology the problem must be specified in a highly structured way that conforms to the program generator's assumptions about software systems.

Our study does not indicate a trend to a particular flavor of methodology. Methodologies with different flavors are needed for different application areas and organizations. For example, a problem with well-defined requirements in a well-understood application area might allow the use of automatic generation techniques while an ill-defined problem might be better attacked by a process-oriented methodology using rapid prototyping.

5.3.3.2. Relation to tools

In addition to their flavors, methodologies can be characterized by their relation to tools:

(1) *Tool centered.* A tool-centered methodology is built around the capabilities and limitations of a set of software tools.

(2) *Tool independent.* Although tools may be used to support the methodology, a tool-independent methodology is not intimately bound to the features of a specific set of tools.

Early methodologies and tools tended to be developed and applied in an ad hoc manner. As an outgrowth of these experiences, the relationship between methodologies and tools is better understood. Both methodologies and tools are being placed on a better formal basis. Future methodologies will rely more heavily on tool support than current methodologies do, but will seek to be independent of particular tool implementations.

5.3.4. Architecture

Most current programming environments are built upon a centralized collection of computing resources (e.g.,a DEC VAX minicomputer installation) with low-resolution display terminals placed near the users' work locations. Recent trends in computer architecture and networking have brought other hardware organizations to the fore.

5.3.4.1. Workstations and servers

The changes in the technology and economics of computing in recent years has *made the dedication of computing resources to individual users or specialized tasks* feasible. Moreover, the technology allows the hardware to be moved out of the controlled environment of the computer room into the users' work areas. These distributed computing elements can share resources and exchange information via high speed networks.

Two types of elements are of interest: workstations and servers. The term workstation is generally applied to a collection of resources, including one or more processors, dedicated to one user at a time. A server is a network node with a specialized function such as operating printers or maintaining a common file system. For our purposes we will classify workstations and servers into two categories:

(1) *Low cost.* A low-cost workstation (or server) might be built around a standard personal computer system with specialized networking hardware and software—putting additional computing power into the hands of the user without incurring large up-front costs. This inexpensive type of workstation can either be connected to a host system as an interactive terminal or to a network of other personal computers, servers, and hosts.

(2) *High cost.* The workstation products market is probably the most intense area of competition in the computer industry today. These workstations are typically built around a "super-microcomputer" and integrate a good graphics display system. Some of the products are specialized to specific applications (e.g., computer-aided design, publishing) or specific language

environments (e.g., Lisp, Ada®); others are more general in their applicability.

5.3.4.2. Displays

Display systems have been an area of intense development in recent years. High-resolution graphics display systems are now readily available and are becoming economical for applications which heretofore could not justify expensive graphics devices. The availability of low-cost, high-resolution laserprinters has also made high quality hardcopy more economical.

5.3.5. Software Development Environments

A software development environment is a *coordinated collection* of computers and software tools dedicated to supporting the development and maintenance of computer software. A "coordinated collection" means that a development environment must be based upon a set of organizational and operational principles. In our study of the technological trends we use two classification criteria for software development environments: orientation and style.

5.3.5.1. Orientation

We have identified several different orientations or organizing principles for software development environments:

(1) *Incongruous.* An incongruous development environment does not have a single organizing principle; it consists of an unorganized collection of hardware, software, and methodologies.

(2) *Operating system based.* A development environment may be based on the concepts and features of a particular operating system. For example, the UNIX® operating system and its tool set are often touted as an excellent software development environment.

(3) *Language based.* A language-based environment concentrates its support on software written in a specific language, e.g., Ada® or Lisp. The environment's tools and techniques can exploit the characteristics of the supported language.

(4) *Tool based.* A tool-based environment is built around the concepts and capabilities of a key tool. For example, the PSL/PSA package and its underlying relational model could form the basis of such a software development environment.

(5) *Process model based.* The central concept of this class of environments is a model of the series of *activities* required to create or modify a software

system. Using the model, the environment can guide the software developer through the necessary steps, checking that the actions taken at each step are correct.

(6) *Specification model based.* This type of software development environment is organized around a formal semantic model of the *objects* created during the development or maintenance of a software system and the relationships among these objects. The environment can syntactically and semantically validate each object and cross-check it with related objects.

Examples of the first four types of environments exist; these types will continue to be important. Although no truly sophisticated process or specification model based systems have been built, we believe that environments with underlying formal models will gradually replace the more ad hoc approaches.

5.3.5.2. Style

The organizational and usage style of software development environments can be classified along two scales:

(1) *Structured or unstructured.* A development environment with a structured style channels development activities into a well-ordered progression through the development process. It may enforce (or restrict) the usage of specific tools for key activities. For example, the programmer may be prevented from entering a coded program module until the design for the module has be checked and approved. A development environment with an unstructured style does not automatically enforce a structured usage of the tools.

(2) *Closed or open.* A closed software development has a fixed set of concepts and tools that can be used by software developers; no other tools are allowed. In an open system, developers are free to introduce new tools and tailor existing tools to better meet the needs of a particular project (or perhaps to better suit their working style.)

5.4. MPE EVOLUTIONARY PATH

In this section we give our assessment of the current status of the DMA MPE and recommend a path for future development. We make recommendations in the five technological areas on which we are focusing in this study. For the convenience of the reader we present this information in a tabular format. We use the taxonomy and terminology presented in the previous section.

5.4.1. Tools

The DMA MPE is a good implementation of an industrial programming environment. Most of the MPE tools are typical of the ones that have been recently introduced into the software development practices of the defense industry. The basic thrusts of our recommendations are (1) to increase the capabilities of the MPE tools by gradually placing them on a better formal foundation, (2) to expand the tool set in a modular, methodology-independent way, and (3) to emphasize development of tools to support maintenance activities.

CURRENT STATUS
(Tools)

CAPABILITY:	syntactic processing
LIFE-CYCLE INTEGRATION:	partial traceability
METHODOLOGICAL SUPPORT:	independent of any methodology
APPLICABILITY:	new development and maintenance (by retrofitting)
OBJECTIVE:	synthesis and limited analysis
LIFE-CYCLE PHASE:	all phases

RECOMMENDATIONS
(Tools)

CAPABILITY:

- strengthen syntactic processing
- include limited semantic checking

LIFE-CYCLE INTEGRATION:

- automate bidirectional traceability
- emphasize phase-wise substitution of tools
- stress ability to integrate tools
- accept parallel methodologies

METHODOLOGICAL SUPPORT:

- emphasize methodology independence

APPLICABILITY:

- strengthen the maintenance component
- acquire retro-documenting support tools (to support maintenance of old programs)

OBJECTIVE:

- expand analytical capabilities in general
- expand analytical capabilities for old programs

LIFE-CYCLE PHASE:

- no recommendations

5.4.2. Human Interfaces

In the area of human interfaces we recommend that DMA evolve the current MPE by exploiting the new graphics technologies and pursuing tools which allow more dynamic interaction and structured information access.

CURRENT STATUS
(Human Interfaces)

STYLE OF INTERACTION:	linguistic
MODE OF INTERACTION:	static (examine results)
ACCESSIBILITY TO INFORMATION:	localized

RECOMMENDATIONS
(Human Interfaces)

STYLE OF INTERACTION:

- improve graphic output capabilities
- move to graphic/linguistic editing

MODE OF INTERACTION:

- attempt to acquire some dynamic capability

ACCESSIBILITY TO INFORMATION:

- move toward structured access

5.4.3. Methodology

We recommend that DMA continue to add new tools (especially for measurement and evaluation) and to refine it's three-tier process-oriented methodological approach to software development and maintenance.

CURRENT STATUS
(Methodology)

FLAVOR: (three tier approach)
Management Methodology—process-oriented
Production Scenarios—process-oriented
Software Engineering Methodologies—specification-oriented

RELATION TO TOOLS:
global tool independence
customized software engineering methodologies

RECOMMENDATIONS
(Methodology)

Current approach assures easy extension to maintenance of GDP systems software

FLAVOR:

- add some process-oriented capabilities at the Software Engineering level
- further integrate the three tier approach (tool interfacing, measurement and evaluation)
- tune the maintenance production scenarios (using measurement and evaluation)

RELATION TO TOOLS:

- add new tools

5.4.4. Architecture

We recommend that DMA keep up with new developments in architecture by using up-to-date display systems and by distributing processing out from the host to low-cost workstations. However, DMA should maintain centralized control over the data needed to manage the software projects.

CURRENT STATUS
(Architecture)

WORKSTATIONS AND SERVERS:	low power and limited role
DISPLAYS:	limited capability

RECOMMENDATIONS
(Architecture)

WORKSTATIONS AND SERVERS:

- distribute processing to workstations
- centralize management data
- seek low-cost workstations

DISPLAYS:

- keep up with the market

5.4.5. Software Development Environments

To meet the needs of DMA in the area of measurement and evaluation, we recommend that DMA formalize it's MPE design by evolving it toward a more structured, process model based approach.

CURRENT STATUS
(Software Development Environments)

ORIENTATION: incongruous

STYLE: structured (regarding SDDL usage)
open (IS/Workbench)

RECOMMENDATIONS
(Software Development Environments)

ORIENTATION:

- stress a process model based approach (supporting of measurement and evaluation)

STYLE:

- maintain the modular structured style for the main line of development activities
- encourage an open style in the low level technical activities

5.5. CONCLUSION

Out attempt to chart a course for the future of the MPE Facility led us to a greater understanding of software development environments in general. The result is a comprehensive characterization and classification of software development environment issues, not available to date in the open literature. Applied to the specific case of the MPE Facility, this taxonomy proved to be an invaluable conceptual tool which enhanced the objectivity of our study.

APPENDIX A

Annotated Bibliography

Authors: Casto, T. L., and Bell, D. M.

Title: Universal Rectifier System Study

Source: Harris Corporation

Abstract: This report documents the results of the Digital Rectification Architecture study. The study encompasses two main thrusts. The first examined suitable architectures for performing two-dimensional rectification on large images at a very fast rate. The second examined system level trade-offs and recommended a total system configuration. The rectification process requires large scale storage (600M pixels), fast random access to the entire image (up to 64M pixels/sec), and multiple arithmetic operations on each pixel processed. The functional requirements imposed by the memory system and pixel processing are the crux of this study.

Authors: Claire, Robert W., and Guptill, Stephen L.

Title: Spatial Data Structures for Selected Data Structures

Source: Proceedings of Auto-Carto 5, American Society of Photogrammetry, 1983

Abstract: The discipline of mathematics is characterized by a small, fundamental set of operators which, when taken in concert, support complex manipulations and analyses. In a similar sense, higher levels of spatial applications programming are founded upon a fundamental set of spatial operators. The tendency to focus upon geometric elements as operators based upon the dimensionality of space is perhaps due to the straightforward manipulations of these elements on a conceptual level. Given the necessity for continuous geometric elements to conform to a discrete, sequential computer storage, our purview should be extended to a primal level that exploits the discrete qualities of spatial data representations.

Author: Clarkson College

Title: Final System Specification for the Clustered Cartographic Processor

Source: Electrical and Computer Engineering Department, Clarkson College, Potsdam, NY

Abstract: This report consists of a functional overview of the Clustered Cartographic Processor System (CCPS). The CCPS is a system which has been designed to: receive cartographic data from a variety of encoding devices; transform that data into a common form; edit it; and prepare a final output file in DMA Standard Planimetric Format. The system will ensure that all data produced is a faithful representation of data originally entered. This document is intended to provide a knowledge of how data flows through the system; what the system components are and how they interact; and to give a brief description of the processes utilized within the system. Each of these areas is given a section of the report as follows: the functional flow, the functional requirements, the system configuration, an operating scenario, the data structures, the assumptions used as a basis for the design, and several simulation results.

Author: DMAAC

Title: Requirement Specification for a Computer Assisted Photo Interpretation System (CAPI)

Source: Aerospace Cartography Department, DMAAC

Abstract: This specification defines and describes the requirements for an analytical stereophotogrammetric, feature-extraction/digital-compilation referred to as the Computer-Assisted Photo Interpretation (CAPI) system. This document specifies the minimum functional system requirements which are considered by the Defense Mapping Agency to be necessary to improve operational efficiency and product quality to satisfy production requirements.

Author: DMAHTC

Title: Statement of Work for Terrain Edit System/Elevation Matrix Processing System (TES/EMPS)

Source: Technology Office, Department of Topography, DMAHTC

Abstract: This statement of work is part of an RFP for the Terrain Edit System/Elevation Matrix Processing System for use at DMAHTC. The TES/EMPS system described in this document deals with the editing and analysis of terrain elevation matrices maintained by DMAHTC. The primary uses are for the correction of digital elevation matrices (DEM) after generation from maps and stereophotographs, and the merging of new data with existing data in hole file, paneling, and feathering processes. In addition, TES/EMPS will be used to maintain archives of DEM data, coverage information, and data quality information (using the figure of merit added to the upgraded UNAMACE).

Authors: Driver, Beth, Armbruster, Sue, Liles, William, Nugent, Ed, Rossoto, Tom, and Serina, Nina

Title: Requirements Analysis for Phase II DBS

Source: Technology Service Corporation

Abstract: This Requirements Analysis for the Data Base System of the DMA Phase II Computer Replacement Program (DAEA18-74-A-0010) is written to provide: (1) the data base system requirements which will serve as a basis for mutual understanding between the user and the developer; (2) information on the data to be stored within the Phase II system and the various ways the data is used; (3) information on the quantity of data stored, processing loads, performance requirements, preliminary design philosophy, and user impacts; (4) information related to priorities for phased implementation of the Phase II system; and (5) a basis for the development of system tests.

Authors: Drobot, Frank N., Heilman, Robert, Melton, Michael, Myers, Louis B., and Went, Burton H.

Title: Study of Digital Elevation Data Processing at DMAHTC

Source: Battelle, Columbus Laboratories, Tactical Technology Center, 505 King Avenue, Columbus, OH

Abstract: At the request of the Defense Mapping Agency, Batelle-Columbus has conducted a study of the processes by which digital elevation matrices (DEM's) are produced. Estimates of current processing costs have been developed, together with a set of functional requirements for a computer processing system. New techniques for interactively editing digital elevation data have also been suggested.

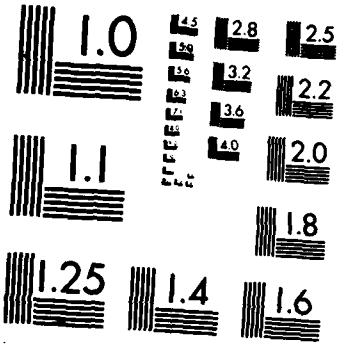
Authors: Hillingworth, Donna K., Lan, Min-Tsung, Liles, William C., Lloyd, James E., and Nugent, Edward D.

Title: Data Management System Strategy on Associative Array Processor

Source: Technology Service Corporation

Abstract: This report documents research efforts in developing a data management system strategy for large cartographic data bases. The effort used an associative array processor equipped with a high-bandwidth mass memory as a back-end processor. The final report is divided into three parts. The first details the major components of the problem: data management, relational data bases, the associative processor STARAN, relational algebraic operators and their algorithms, DMA data files, the Digital Landmass

AD-A173 599 LONG-RANGE TECHNOLOGICAL IMPACT ON COMPUTER-AIDED 3/3
PRODUCT DEVELOPMENT AT (U) WASHINGTON UNIV ST LOUIS MO
DEPT OF COMPUTER SCIENCE G ROMAN ET AL JUL 86
UNCLASSIFIED RADC-TR-86-49 F30602-83-K-0065 F/G 9/2 NL


1.0
4.5
5.0
5.6
6.3
2.8
2.5
3.2
3.6
2.2
4.0
2.0
1.1
1.8
1.25
1.4
1.6

System (DLMS) elements and their conversion to a relational format, and a possible data base machine. The second part details the pilot-system implementation on the STARAN, including a user's manual and a programmer's manual describing the query language and operations available on the pilot system as well a listings and documentation. The third part contains a system functional description and user's manual, detailing the features and functions a full-scale system must have.

Author: Mirkay, Francis M.

Title: Defense Mapping Agency Large Scale Data Base Integration

Source: Proceedings of Auto-Carto 5, American Society of Photogrammetry, 1983

Abstract: The Defense Mapping Agency has massive amounts of MC&G digital data holdings resulting from programs to support DoD advanced weapons systems and automated map and chart production. The digital data is used to support the DoD and other countries as well as internal DMA production. DMA requires a digital data base to maintain, store, retrieve, compare, and report on this MC&G digital data. This paper addresses current DMA efforts that will evolve into an on-line interactive, distributed, networked data base system and its associated environment for data handled by HQ DMA and the DMA Production Centers.

Author: PRC Information Sciences Company

Title: Clustered Carto Processing System: Final System Design

Source: PRC Information Sciences Company, 1500 Planning Research Drive, McLean, VA

Abstract: The final Clustered Carto Processing System (CPS) design presented in this document is intended to: (1) provide the Government with an understanding of the CPS design with enough detail for subsequent approval of the approach and direction taken; (2) provide the system developers with a final design to support subsequent specification efforts; and (3) provide a design baseline from which system modeling and timing analysis can verify that the system will meet or exceed the stated performance requirements. The effort to produce the final design follows a previously developed in-progress design and Government review and comment on the design. Both the in-progress and final design followed an analysis and validation of the requirements as presented in the Statement of Work for the CPS (a separate document). These requirements were supplemented with underlying design concepts by the given Preliminary System Design and Preliminary Software Specifications contained in the Statement of Work. This document presents general and detailed design of the CPS; a description of the system modeling and timing analysis for the CPS; results of the review and analysis of the given design and software specifications; and the requirements traceability which provides a linkage between the CPS requirements and design.

APPENDIX B

REQUIREMENTS VALIDATION CASE STUDY

1. INTRODUCTION

This appendix describes a case study in which the formalism is applied to a number of simple GDP problems. The formal problem descriptions, taken together, constitute the functional requirements for a hypothetical GDP system called MAPX. We have two reasons for developing the case study: we wish to show that the formalism can be implemented, and to identify the key technical difficulties involved in a realistic implementation of the formalism for use in a GDP requirements engineering environment. A requirements engineering environment is a collection of tools for defining data and processing requirements for a particular class of applications. One of the most important services provided by a requirements engineering environment is the ability to simulate the operation of a proposed system based on its requirements. Ideally, the simulation should require only formal requirements and test data; it should not involve the design of a prototype. The advantage of requirements simulation is that it allows developers and users to discover errors in requirements before any resources are invested in design. The effect is to reduce the risk associated with system development by decreasing the probability that design effort will be wasted because of a misunderstanding of requirements. Design costs of GDP systems are often high, so the economic benefits of requirements simulation should be readily apparent.

The case study consists of a simulator, written in Prolog, data requirements for several simple example problems, and test data. Processing requirements are limited to queries over existing data, and both data requirements and queries are expressed in Prolog. The simulator can provide both textual and graphic responses to queries. Prolog was chosen as the implementation language because of its ease of expressing logic. The example problems were chosen to illustrate various parts of the formalism, and include hole detection, spike detection, accuracy assignment, map production, and road path layout. Section 2 details the Prolog representation of facts and semantic constraints, and Section 3 discusses the data requirements for each sample problem, and the output generated by the simulator.

The major problems we encountered in developing the simulator were the lack of efficient primitives for reasoning about spatial relationships, and the inefficiency of Prolog when processing large volumes of data. Ideally, any language for simulating GDP requirements should contain a set of primitives for reasoning about spatial relationships between geographic entities (e.g. distance between two entities and inclusion of one entity in another). In our implementation, we found it necessary to write (in Prolog) a number of spatial operations because Prolog does not directly support spatial logic. We developed only the operations needed for the case study; we did not attempt to construct a full set.

The inefficiency of Prolog for processing large volumes of data presented a more serious problem than the lack of spatial operations. Geographic information often covers large areas represented by thousands of points, and several facts may be required to describe each point. For example, elevation data for a one-square-mile area with one thousand sample points per mile requires one million basic facts. The ability to process this much information efficiently is far beyond the capacity of current Prolog implementations. Although Prolog can store the facts in its internal database, it may take hours to process even a simple query. Our solution was to store high-volume spatially qualified information in files separate from the Prolog database, and to add a number of language primitives for accessing the files. The files and primitives used to store and manipulate high-volume information are described in Section 4.

2. SYSTEM OVERVIFW

The Case Study requirements simulator consists of a set of Prolog routines which interpret formal GDP data requirements, and two support routines for generating graphic output. A number of C subroutines for spatial reasoning, described in Section 3, were made available as Prolog primitives. Requirements initially expressed in the formalism described in Section 3.5 are translated to Prolog for execution. We choose to do the translation manually, although it can be done automatically. The following features of the formalism are supported by our implementation:

- Basic and virtual facts
- Spatial qualification
- Temporal qualification
- Accuracy
- Models of reasoning

The implementation can produce textual and graphic responses to queries. Textual responses are generated directly by the Prolog interpreter, and graphic displays are produced from files of spatially qualified information stored separately from the Prolog database. An overview of the implementation structure appears in Figure 3.6-1. Below is a brief description of each component.

Low-Volume Sample Data: Contains low-volume basic facts.

Prolog Interpreter: Executes processing requirements (expressed as user queries) using sample data and data requirements.

Formalized Data Requirements: Data requirements (virtual facts and meta-facts) originally expressed in our formalism and then translated into Prolog.

High-Volume Sample Data: Contains high-volume spatially qualified information. Mapx Matrix Files represent collections of basic facts which describe the points in a rectangular region. Each basic fact associates a value from a semantic domain (e.g. elevation or vegetation type) with a point in the region.

Mapx File Operations: Are used by the Prolog interpreter and Imager to access High-Volume Sample Data. These operations are described in detain in Section 3.

Imager: Produces 3-channel color images from Mapx Matrix Files by mapping semantic domain values onto color intensities.

Overlay Program: Superimposes one image onto another. For example, in one of the sample problems, a road path is overlaid onto an elevation image.

It should be noted that Formalized Data Requirements and Low-Volume Sample Data

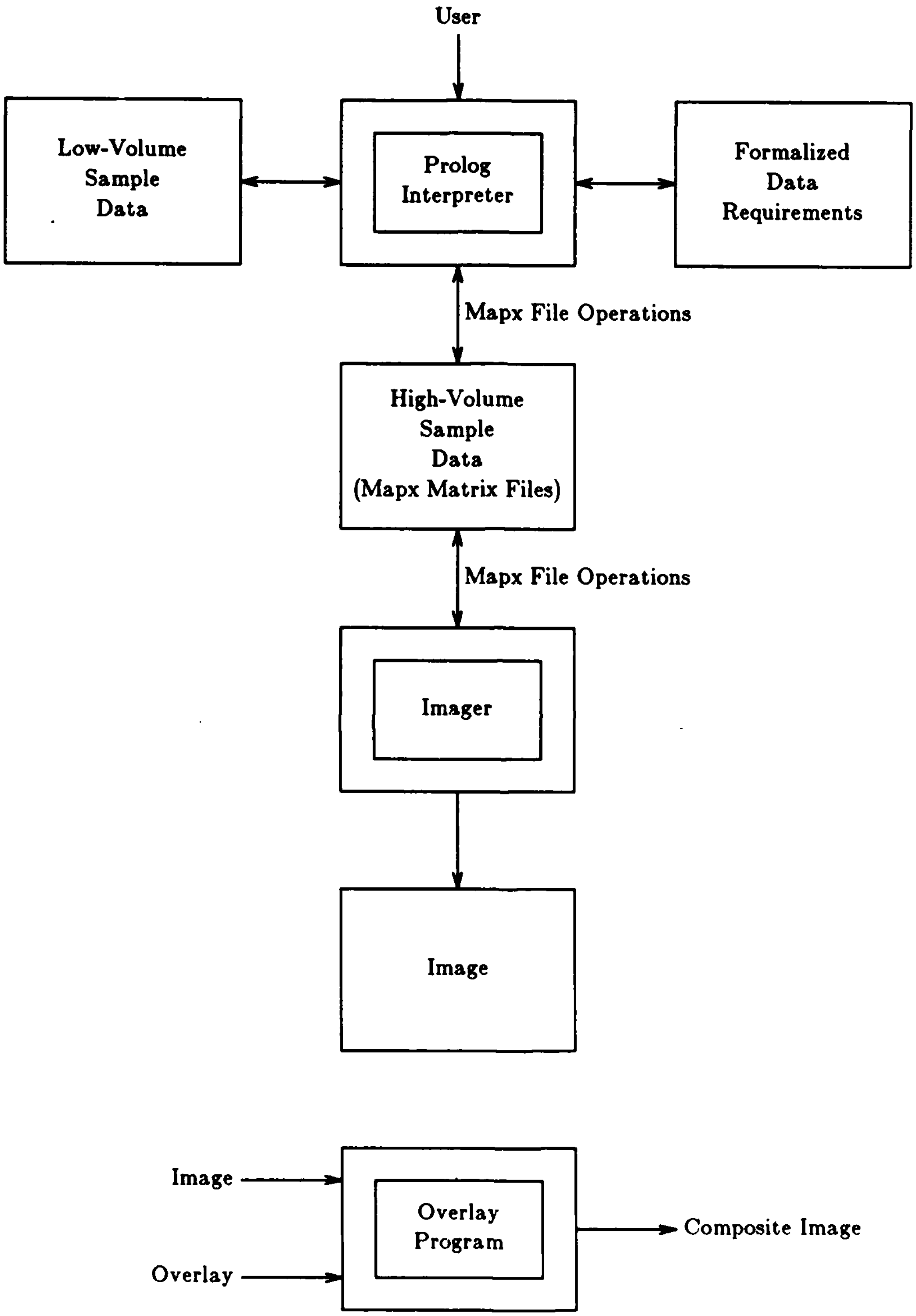

User
Low-Volume Sample Data
Prolog Interpreter
Formalized Data Requirements
Mapx File Operations
High-Volume Sample Data (Mapx Matrix Files)
Mapx File Operations
Imager
Image
Image
Overlay
Overlay Program
Composite Image

Figure B-1

are both stored in the Prolog Interpreter's internal database and are indistinguishable from the interpreter's viewpoint. We distinguish them on the diagram because they are used differently: one expresses system requirements and the other is used in testing the requirements.

3. MODELLING STRATEGY

This section describes our use of the Prolog language [CLOC81] to interpret the formalism. Prolog programs consist of a collection of *facts* and *rules*. *Structures* are used to represent real-world entities, and to compose facts and rules, which correspond to statements in predicate calculus representing knowledge about the real world. (Our terminology differs somewhat from that of [CLOC81].) Prolog programs are thus ***descriptive*** in that the collection of Prolog facts and rules represents knowledge of the world. By contrast, programs written in more traditional languages, such as FORTRAN and Pascal, are ***prescriptive*** in that they consist of a sequence of imperative commands. The purpose of the Case Study simulator is to interpret a formalism, based on logic, for describing data requirements, so Prolog is a natural choice for the implementation language.

Prolog structures may be recursively defined as

> a structure name, called a *functor,* optionally followed by a parenthesized sequence of structures (parameters), or
>
> a number, or
>
> a list of structures enclosed in square brackets.

Prolog facts consist of a structure of the functor-parameter form followed by a period. Prolog facts are analogous to basic facts in our formalism: they are assumed to be true without regard to other facts. (It should be noted, however, that our basic facts have a much narrower use than Prolog facts. Basic facts are restricted to geographic entities while Prolog facts may refer to anything.) For example, we could represent the fact that a person named John owns a book called "The Tale of Two Cities" by the writing

```
owns(john,book(tale_of_two_cities)).
```

If John also owns a television, we could add

```
owns(john,television).
```

Prolog rules represent statements in predicate calculus, the truth of which depends on the truth of facts and other rules. Thus, rules correspond to virtual facts in our formalism. Rules consist of a structure of the form functor-parameters, an implication operator, and a sequence of functor-parameter structures. For example, the statement

"John may use a thing if he owns it or it is owned by a friend"

can be represented in predicate calculus as (for-all F, X) ((owns(john,X) ¦ (owns(F,X) ^ friend(john,F))) → may_use(john,X))

and in Prolog as

```
may_use(john,X) :- owns(john,X); (friend(john,F), owns(F,X)).
```

In Prolog, implication is denoted by ":-", conjunction by the comma, and disjunction by the semicolon. Numbers and lower case alphanumeric strings represent values, and variables are represented by capitalized strings. (We use the convention of writing variables in upper case.) All variables in Prolog are universally quantified.

We use Prolog facts to represent basic facts as shown:

```
fact(PRED,model(M),prob(P),loc([X,Y]),date([MO,YR]),security(S),
        [P1, P2, ..., Pn], [O1, O2, ..., Om], STATUS).
```

where

PRED is the fact name (predicate). For example, elevation. It may seem more natural to use the fact name as the functor, but this would prevent implementation of meta-facts, in which predicates are universally quantified.

M is the name of a model for grouping facts.

P is a real number between 0 and 1 inclusive, and represents qualification by accuracy.

X and Y are Cartesian coordinates of a point in absolute space representing spatial qualification. (Support of the area uniform, area averaged, and area sampled operators is discussed below.)

S represents a security classification. (Security is present for consistency with the formalism but is not used in the case study.)

P1, P2, ..., Pn ($n \geq 0$) represent semantic domain values (e.g. temperature or feet above sea level).

O1, O2, ..., Om ($m \geq 1$) are object designators.

STATUS indicates the availability of the fact for use by virtual facts and meta-facts. In the current implementation, this field always contains the value "asserted".

For example, a collection of elevation samples could be expressed in Prolog by:

```
fact(elev,model(_),prob(_),loc([100,100]),date(_),security(_),[250],[land],asserted).
fact(elev,model(_),prob(_),loc([100,101]),date(_),security(_),[249],[land],asserted).
fact(elev,model(_),prob(_),loc([101,100]),date(_),security(_),[253],[land],asserted).
fact(elev,model(_),prob(_),loc([101,101]),date(_),security(_),[248],[land],asserted).
```

Expressed in the formalism,

@ [100,100] elev (250) (land)
@ [100,101] elev (249) (land)
@ [101,100] elev (253) (land)
@ [101,101] elev (248) (land)

Even though the basic facts listed above are qualified only by space, the Prolog representation requires that all parameters be included, in the proper order. This is necessary because of the way in which Prolog searches its database to answer queries. The underscore character ("_") is used as a placeholder and indicates that the value which would appear in its place is irrelevant.

Spatial qualification in the example above is by the simple spatial operator. We can represent the area uniform, area averaged, and area sampled operators by replacing loc(X,Y) with one of the following:

loc(uniform(R),[X,Y]) meaning $@_u$ [R] (X,Y) ...
loc(averaged(R),[X,Y])meaning $@_a$ [R] (X,Y) ...
loc(sampled(R),[X,Y]) meaning $@_s$ [R] (X,Y) ...

R is a symbol (or a variable instantiated to a symbol) representing a resolution function which relates points in some logical space to points in absolute space. We have imposed the restriction of allowing queries and virtual facts to reference at most one level of resolution. Thus, virtual facts and queries may not reference two different logical spaces or a logical space and absolute space. It is possible to relate various levels of resolution using meta-facts, but this was not required for any of the sample problems.

Virtual facts are written as Prolog rules, the header of which have the same form as basic facts. For example, a virtual fact defining average elevation for a 3 by 3 area could be written:

```
fact(avg_elev,model(_),prob(_),loc([X,Y]),date(_),security(_),[Z],[land],asserted) :-
        XLL is X - 1,
        YLL is Y - 1,
        XUR is X + 1,
        YUR is Y + 1,
        fact(elev,model(_),prob(_),loc([XLL,YLL]),date(_),security(_),[Z1],[land],asserted).
        fact(elev,model(_),prob(_),loc([XLL,Y]),date(_),security(_),[Z2],[land],asserted).
        . . .
        fact(elev,model(_),prob(_),loc([XUR,Y]),date(_),security(_),[Z8],[land],asserted).
        fact(elev,model(_),prob(_),loc([XUR,YUR]),date(_),security(_),[Z9],[land],asserted).
        Z is (Z1 + Z2 + Z3 + Z4 + Z5 + Z6 + Z7 + Z8 + Z9) / 9.
```

The "is" operator is a Prolog primitive allowing evaluation of arithmetic expressions. The variables XLL, YLL, XUR, YUR respectively are coordinates of the lower left and upper right corners of the the 3 by 3 area centered at (X,Y). The representation of virtual facts makes them indistinguishable from basic facts when referenced by an expression or query. This is consistent with the formalism.

Semantic constraints have the same form as virtual facts, except that the functor "fact" is replaced with "error". For example, it is an error to have two distinct elevation values for the same point. We can express this in Prolog as:

```
error(elev_conflict,model(_),prob(_),loc([X,Y]),date(_),security(_),[ ],[land],asserted) :-
        fact(elev,model(_),prob(_),loc([X,Y]),date(_),security(_),[Z1],[land],asserted),
        fact(elev,model(_),prob(_),loc([X,Y]),date(_),security(_),[Z2],[land],asserted),
        Z1 != Z2.
```

Although semantic constraints are just a special case of virtual facts, we distinguish them because they have a different purpose. Virtual facts are general rules by which we derive new knowledge from what we already know; semantic constraints indicate errors in existing information.

Meta-facts and meta-constraints have the same form as virtual facts and semantic constraints, except that the predicate is universally quantified (i.e. a variable). For example, we can generalize the above constraint to include any single-valued property by stating it as a meta-constraint:

```
error(ERRORTYPE,model(_),prob(_),loc([X,Y]),date(_),security(_),[],OBJECT,asserted) :-
        singlevalued(PRED),
        fact(PRED,model(_),prob(_),loc([X,Y]),date(_),security(_),PARM1,OBJECT,asserted),
        fact(PRED,model(_),prob(_),loc([X,Y]),date(_),security(_),PARM2,OBJECT,asserted),
        PARM1 != PARM2,
        errorname(PRED,"_conflict",ERRORTYPE).
```

The variable PRED is an arbitrary predicate (fact name). The property "singlevalued(PRED)" states that the parameters of the fact PRED may have at most one value for a given object and point in space. The function "errorname" returns a unique constraint name based on the predicate. In the case of elevation (elev), the name of the constraint would be "elev_conflict".

4. SAMPLE PROBLEMS

This section describes the six sample problems which comprise the case study. The following problems are presented:

- Hole Detection
- Spike Detection
- Accuracy Assignment to Ocean Depth Values
- Temporal Reasoning (Continuity Assumption)
- Map Production
- Road Path Layout

The discussion of each sample problem consists of an informal description of the problem to be solved, a definition of the data requirements in the formalism of Section 3.5, and the Prolog code used by the simulator.

4.1. Hole Detection

Informal Description: Holes are points on land for which no elevation is known and none can be derived. Such points are considered to be errors because they represent incomplete information. Hole detection is a quality assurance procedure commonly applied to new elevation samples before they are made available for general use. It is natural to define a hole negatively as a point for which there is no fact (basic or virtual) stating the elevation. A problem with this type of definition is that it is satisfied by any point outside the sample area. Thus, the set of points satisfying the definition is infinite. The response of Prolog to a query based on a negative definition is an uninstantiated (i. e. undefined) variable. In order to get a meaningful response, we must restrict our attention to a well-defined finite area in logical space. In the illustration, we restrict the query to a rectangular region. Since the region contains a lake, we must define a hole to be a point for which there is no elevation information and no depth information.

It should be noted that the "not" operator, both in the formal definition and in Prolog, does not mean logical negation. Rather, it means that the truth of the statement in its scope cannot be established given the information available. By the ***open world assumption,*** discussed in Section 3.5, such statements are not automatically assumed to be false.

Formal Definition:

$$\begin{array}{l}
\text{(for-all X, Y, Z1, Z2)} \\
\qquad \text{(inside(point(X,Y),rectangle(191,182,341,332)) AND} \\
\qquad \text{ NOT (}@_a\text{[r] [X,Y] input'elev(Z1) (land)) AND} \\
\qquad \text{ NOT (}@_a\text{[r] [X,Y] depth(Z2) (ocean))} \\
\qquad\qquad \rightarrow @_u\text{[r] [X,Y] input'error (hole) (land))}
\end{array}$$

Prolog Definition:

```
error(hole,model(input),_,loc(uniform(r),[X,Y]),_,_,_,[land],asserted) :-
    inside(point([X,Y]),rectangle([191,182,341,332])),
    not(fact(elev,model(input),_,loc(averaged(r),[X,Y]),_,_,[Z1],[land],asserted)),
    not(fact(depth,model(_),_,loc(averaged(r),[X,Y]),_,_,[Z2],[ocean],asserted)).
```

4.2. Spike Detection

Informal Description: Spikes are points of sudden increase in elevation. They are typically caused by failures in remote sensing equipment (e.g., radar altimeters) when collecting elevation samples. Spikes are errors of the same type as holes (points for which there is no known elevation), and most often occur in new elevation samples. Spike detection is a quality assurance procedure typically applied, along with hole detection, before new elevation samples are made available for general use. Spikes are somewhat harder to define than holes: there is an elevation value, but it may be incorrect. It is possible, especially with low resolution samples, for elevation to suddenly increase at one point and still be correct. Deciding which elevation values are in error often requires knowledge of the area obtained independently of the elevation sample. For this reason, it is natural to define spikes with fuzzy constraints. Points with the greatest probability of having incorrect elevation deserve the most attention. We have chosen a simple definition of the probability of error: the ratio of increase between a single point and the average of its neighbors.

Formal Definition:

(for-all X Y Z ZA A)
($@_a[r]$ [X,Y] input'elev (Z) (land) AND
ZA = avg_elev (land, r, rectangle(X−1, Y−1, X+1, Y+1)) AND
$Z > ZA$ AND
$Z \neq 0$ AND
$A = (Z - ZA) / Z \rightarrow @_u[r]$ [X,Y] %A input'error (spike))

Prolog Definition:

```
error(spike,model(input),prob(A),loc(uniform(r),[X,Y]),_,_,_,[land],asserted) :-
    fact(elev,model(input),_,loc(averaged(r),[X,Y]),_,_,[Z],[land],asserted),
    XL is X-1,
    YL is Y-1,
    XH is X+1,
    YH is Y+1,
    avg_elev(land,r,rectangle([XL,YL,XH,YH]),ZA),
    Z > ZA,
    Z != 0,
    ZDIFF is Z - ZA,
    A is ZDIFF / Z.
```

4.3. Accuracy Assignment to Ocean Depth Values

Informal Description: Ocean depth values can be obtained from sonar readings. A ship with a sonar apparatus travels over an area of ocean recording the depth values for points directly beneath. The path travelled by the ship does not necessarily consist of straight lines. Points over which the ship passed directly form *contours,* and depth values are completely accurate only for these points. Values for other points are interpolated based on the distance from the closest contour. This interpolation introduces the possibility of error. (We disregard the possibility of inaccuracy in the sonar equipment.) Thus, it is necessary to assign figures of merit to depth values in the same way as to elevation values. It is possible to express confidence in depth values by fuzzy constraints, but for purposes of illustration we choose to assign a probability of correctness to the facts defining depth rather than a probability of error to a fuzzy constraint.

In the descriptions below, the model "qualified" contains a single virtual fact which qualifies depth values by accuracy. The model "input" contains basic facts designating which points lie on contours and the depth for those points, as well as virtual facts which interpolate depth values for other points.

Formal Definition:

```
(for-all X Y Z D A)
        (@[X,Y] input'depth (Z) (ocean) AND
         D = dist_from_contour([X,Y], ocean, input) AND
         A = 1 − min(0.02*D, 1.0) → @[X,Y] %A qualified'depth (Z) (ocean))
```

Prolog Definition:

```
fact(depth,model(qualified),prob(A),loc([X,Y]),_,_,[Z],[ocean],asserted) :-
     fact(depth,model(input),_,loc([X,Y]),_,_,[Z],[ocean],asserted),
     dist_from_contour([X,Y], ocean, input, D),
     D1 is 0.02 * D,
     max([D1, 1.0], D2),
     A is 1.0 - D2.
```

4.4. Temporal Reasoning (Continuity Assumption)

Informal Description: Our illustration of temporal reasoning shows the use of the Continuity Assumption, discussed in Section 3.5. Under the Continuity Assumption, a fact which is qualified by some point in time is considered true for all successive times, until it is overridden by a more recent fact. (In this context, "more recent" refers to the time by which a fact is qualified, not the time at which it was entered into the system. It is easy to conceive of situations in which historic information is recorded in non-chronological order). For example, suppose that a system contains information obtained from ocean shoreline surveys conducted in 1970 and 1980. Both surveys produced coordinates of points on the beach, and the coordinates are recorded as temporally qualified basic facts. Presumably, the shorline location changed somewhat between the two surveys. Without the Continuity Assumption, or some other model of temporal reasoning, the system can answer queries about shorline location only for 1970 or 1980; it will not be able to determine if a given point is on the beach in 1975. However, with the Continuity Assumption, queries for 1970 or any later time can be answered. The shorline location for any time between 1970 and 1980 will be considered to be the same as for 1970; the location for any time after 1980 will be the same as for 1980. It should be noted that the Continuity Assumption is not the only method of temporal reasoning. In this example, it would be more appropriate to assume that changes in shorline location occur gradually rather than instantly. We illustrate the Continuity Assumption because it is one of the most simple models of temporal reasoning.

In the descriptions below, the Continuity Assumption is stated as a meta-fact, and information which is inferred using it is grouped under the model "continuity". In order to avoid a circular definition, facts to which the meta-fact refers must exist under some other model.

Formal Definition:

```
(for-all M P T T′ F O)
    (&T @P M'F(O) AND
     $M \neq$ continuity AND
     $T' > T$ AND
     (for-all P′′ T′′) ($T < T'' \leq T'$ →NOT (&T′′ @P′′ F(O) AND))
            → &T′ @P continuity'F(O))
```

Prolog Definition:

```
fact(F,model(continuity),_,loc(P),date(T1),_,_,[O],asserted) :-
    fact(F,model(M),_,loc(P),date(T),_,_,[O],asserted),
    M != continuity,
    T1 > T,
    all(
        betweendate(T,T1,T2),
          not(fact(F,model(_),_,loc(P2),date(T2),_,_,[O],asserted))
       ).
```

4.5. Map Production

Informal Description: As one example of product generation, we use virtual facts to derive vegetation covering from elevation, and produce a map in which vegetation type is identified by color. The virtual facts are based on the knowledge that different types of vegetation predominate at various levels of elevation. For simplicity, we associate only one vegetation type with each elevation.

The first four expressions in the definitions below are virtual facts which define vegetation zones based on elevation. The last expression translates coordinates in logical space to positions on a map, and assigns a color to each vegetation type.

Formal Definition:

```
(for-all P V Z)
        (@$_a$[r] P elev(Z) (land) AND $Z \geq 1035$
              → @$_u$[r] P veg (none) (land) )

(for-all P V Z)
        (@$_a$[r] P elev(Z) (land) AND $750 \leq Z \leq 1034$
              → @$_u$[r] P veg (pine) (land) )

(for-all P V Z)
        (@$_a$[r] P elev(Z) (land) AND $500 \leq Z \leq 749$
              → @$_u$[r] P veg (oak) (land) )
```

(for-all P V Z)
($@_a[r]$ P elev(Z) (land) AND $0 \leq Z \leq 499$)
$\rightarrow @_u[r]$ P veg (grass) (land)

(for-all A C V X Y X′ Y′)
($@_u[r]$ [X,Y] veg(V) (land) AND
(X′, Y′) = map_coord(X, Y, r) AND
C = veg_color(V)
$\rightarrow$ map_color(C, X′, Y′))

Prolog Definition:

```
fact(veg,_,_,loc(uniform(r),P),_,_,[none],[land],asserted) :-
  fact(elev,_,_,loc(average(r),P),_,[Z],[land],asserted),
  Z >= 1035.

fact(veg,_,_,loc(uniform(r),P),_,_,[pine],[land],asserted) :-
  fact(elev,_,_,loc(average(r),P),_,[Z],[land],asserted),
  Z =< 1034,
  Z >= 750.

fact(veg,_,_,loc(uniform(r),P),_,_,[oak],[land],asserted) :-
  fact(elev,_,_,loc(average(r),P),_,[Z],[land],asserted),
  Z =< 749,
  Z >= 500.

fact(veg,_,_,loc(uniform(r),P),_,_,[grass],[land],asserted) :-
  fact(elev,_,_,loc(average(r),P),_,[Z],[land],asserted),
  Z =< 499,
  Z >= 0.

map_color(C, X1, Y1) :-
  fact(veg,_,_,loc(uniform(r),[X,Y]),_,_,[V],[land],asserted),
  map_coord(X1, Y1, X, Y, r),
  veg_color(C, V).
```

5. PROCESSING SPEED

A serious problem we encountered in developing the simulator was the inefficiency of Prolog for processing large amounts of information. We chose to use Prolog for requirements validation because, in addition to being closely compatible with predicate calculus, it is capable of concisely expressing a large class of data structures (see Section 2). Unfortunately, what is gained in generality must be paid for in execution time. The version of Prolog used in the Case Study searches its internal database for (Prolog) facts based on a hash function which uses the functor name and the number of top-level parameters. A linear search is used to choose between patterns which produce the same hash value. Since our representation of basic and virtual facts uses a single functor and a uniform number of parameters, most queries invoked by the simulator reduce to simple linear searches. This is clearly inappropriate for dealing with sample GDP problems which may involve hundreds of thousands of spatially-qualified basic facts. We investigated several alternative ways of encoding basic and virtual facts in Prolog, but found that none had the flexibility of the encoding we currently use.

Over 95 percent of the basic facts used in our examples are spatially qualified. Furthermore, they can be represented by expressions of the form

@P Q (V) (O)

where P is a point in some rectangular region, the semantic domain containing V is a subset of the integers, and O is a geographic object. (An operator for logical space can be used place of the simple spatial operator.) In all cases, the region containing P can be represented by a finite number of points. The predicate Q can thus be viewed as a function mapping a finite set of integer coordinates onto a finite subset of the integers. It was apparent that we could substantially improve execution speed by removing this type of spatially qualified information from the Prolog database, and providing access routines which take advantage of way in which we query spatial information.

Our solution was to store spatial information in files separate from the Prolog database, and to add to Prolog a number of primitives for accessing the files. The files, called *Mapx Matrix Files* represent collections of facts of a single predicate over a rectangularly bounded regions. Each file consists of a header containing coordinates of the lower left and upper right corners of the region, and a data portion of sufficient size to contain one 16-bit value for each point in the region. Values which may be recorded lie within the range -32767 to +32767. The value -32768 is used to indicate that the fact represented by the file is *not* true at that point. We refer to points at which the values are not equal to -32768 as *defined points*. The files are created and accessed by subroutines callable from C and Prolog. The C version of the subroutines is described below—the Prolog version is identical except for minor differences in syntax. Parameters printed in boldface must be defined by the calling routine; values for others are provided by the subroutine.

5.1. MpxPutHeader

Syntax:

MpxPutHeader(**FileName, Header,** FileNumber)

Parameters:

FileName—Name of file to be created
Header—Contains coordinates defining bounding rectangle
FileNumber—Used to reference file by other Mapx routines

Function: A new Mapx file with name given by FileName is created. The header record is stored internally but not written to disk until MpxPutEnd is called. MpxPutHeader must be the first operation on a new Mapx file, and fails if the file already exists. Upon successful completion, FileNumber contains an identifying number to be used when referencing the file with other Mapx file operations.

5.2. MpxPutIPoint

Syntax:

MpxPutIPoint(**FileNumber, XCoord, YCoord, Value**)

Parameters:

FileNumber—Number of a file opened by MpxPutHeader
XCoord—X coordinate
YCoord—Y coordinate
Value—Value to record for point (XCoord, YCoord)

Function: The 16-bit quantity Value is recorded in the given file for the point with coordinates (Xcoord, Ycoord). Values which may be recorded are in the range -32767 to +32767. The quantity -32768 is used to indicate that the fact represented by the file is *not* true at the given point. It is possible to call this routine several times for the same point, in which case only the most recent value is recorded. This operation fails if FileNumber does not reference a Mapx file opened by MpxPutHeader, or if the point lies outside the bounding rectangle given in the file header. Nothing is written to disk until MpxPutEnd is called.

5.3. MpxPutEnd

Syntax:

MpxPutEnd(**FileNumber**)

Parameters:

FileNumber—Number of a file opened by MpxPutHeader

Function: Points recorded by MpxPutIPoint for the file given by FileNumber are written to disk, along with the file header, and the file is closed. Further references to the file must be preceded by MpxInputFile, which opens the file for input operations. This operation fails if FileNumber does not reference a Mapx file currently open for output.

5.4. MpxInputFile

Syntax:

MpxInputFile(**FileName,** FileNumber)

Parameters:

FileName—Name of an existing Mapx Matrix File
FileNumber—Used to reference the file by other Mapx routines

Function: This procedure opens the Mapx Matrix File named by FileName for input, and assigns an identification number. This routine must be called before invoking any input operations, and fails if the given file does not exist or is not a Mapx Matrix File.

5.5. MpxGetHeader

Syntax:

MpxGetHeader(**FileNumber,** Header)

Parameters:

FileNumber—Number of a file opened by MpxInputFile
Header—Contains bounds of rectangular region described by file

Function: The header record for the Mapx Matrix File given by FileNumber is returned. FileNumber must have been assigned by a call to MpxInputFile.

5.6. MpxGetStart

Syntax:

MpxGetStart(**FileNumber,** QueryNumber)

Parameters:

FileNumber—Number of a file opened by MpxInputFile
QueryNumber—Used to distinguish between concurrent queries to file

Function: This procedure initiates a query into a file previously opened by MpxInputFile. Queries sequentially return points from a given area (which may not be the bounding rectangle of the file), the values for which lie within a given range. It is possible to have several concurrent queries in progress for the same file; QueryNumber is necessary to distinguish between them.

5.7. MpxGetIPoint

Syntax:

MpxGetIPoint(**QueryNumber,** XCoord, YCoord, Value)

Parameters:

QueryNumber—Identifies a query initiated by MpxGetStart
XCoord—X Coordinate of next point satisfying query
YCoord—Y Coordinage of next point satisfying query
Value—Value of next point satisfying query

Function: This routine returns the coordinates and value of the next point satisfying the query given by QueryNumber. By default, all defined points are returned, from the lower left corner of the region in the file header to the upper right, with the X coordinate varying most vrapidly. It is possible to restrict the points returned to those within a given subregion of the file (MpxSetRectangle) and with values in a given range (MpxSetVRange). MpxGetIPoint returns an error if it is called after it returns the last point which satisfies the query.

5.8. MpxSetRectangle

Syntax:

MpxSetRectangle(**QueryNumber, XLowerLeft, YLowerLeft, XUpperRight,YUpperRight**)

Parameters:

QueryNumber—Identifies a query initiated by MpxGetStart
XLowerLeft—X coordinate of lower left corner of subregion
YLowerLeft—Y coordinate of lower left corner of subregion
XUpperRight—X coordinate of upper right corner of subregion
YUpperRight—Y coordinate of upper right corner of subregion

Function: This routine limits the set of points returned by MpxGetIPoint on the query given by QueryNumber to those lying within a specific rectangular region. If either of the lower left or upper right corner if the region lies outside the bounding rectangle of the file, the coordinates of that point are adjusted to match the corresponding point in the file's bounding rectangle. By default, MpxGetIPoint returns all defined points. The MpxSetVRange procedure can be used to further limit the set of points to those with parameter values within a specified range.

Associated with each query is a logical pointer which identifies the point to examine on the next call to MpxGetIPoint. MpxSetRectangle sets the logical pointer to the lower left corner of the rectangular region. This routine can be called at any time between when the query is initiated by MpxGetStart and terminated by MpxGetEnd. Thus, a single query can be used to search many regions of a file. MpxSetRectangle returns an error if QueryNumber does not reference an active query, or if the coordinates of the lower left corner of the region exceed those of the upper right corner.

5.9. MpxSetVRange

Syntax:

MpxSetVRange(**QueryNumber, LowValue, HighValue**)

Parameters:

QueryNumber—Identifies a query initiated by MpxGetStart
LowValue—Lower bound of value range
HighValue—Upper bound of value range

Function: This routine limits the set of points returned by MpxGetIPoint for the query given by QueryNumber to those with values within a specified range. By default, MpxGetIPoint will return all points which have defined parameters. The routine MpxSetRectangle can be used to further restrict the query to points within a specific rectangular region. MpxSetVRange returns an error if LowValue is greater than

HighValue, or if either value is outside the range -32767 to +32767.

5.10. MpxExaminePoint

Syntax:

MpxExaminePoint(**FileNumber, XCoord, YCoord,** Value)

Parameters:

FileNumber—Number of a file opened by MpxInputFile
XCoord—X coordinate of point to return
YCoord—Y coordinate of point to return
Value—Value of point at (XCoord, YCoord)

Function: This routine returns the value associated with the point (XCoord, YCoord) for the file given by FileNumber. If the point has no value (-32768 recorded in file), or if FileNumber does not reference a file opened by MpxInputFile, an error is returned.

5.11. MpxClosest

Syntax:

MpxClosest(**FileNumber, X, Y, DistInit, DistInc, DistFinal,** X, Y, V, Dist)

Parameters:

FileNumber—Number of a file opened by MpxInputFile
X—X coordinate of reference point
Y—Y coordinate of reference point
DistInit—Initial distance from reference point to edges of search rectangle
DistInc—Increment to distance from reference point to edges of search rectangle
DistFinal—Maximum distance from reference point to edges of search rectangle
X—X coordinate of point returned
Y—Y coordinate of point returned
V—Value of point returned
Dist—Distance from point returned to reference point

Function: This routine returns the coordinates and value of the defined point which is closest to a given reference point, and which lies within a square region of given size. The parameters DistInit, DistInc, and DistFinal are provided to optimize the search. Initially, a square region the edges of which are DistInit units from the reference point is searched. If no defined point is found, the next area searched is square region with edges DistInit + DistInc units from the reference point, excluding the area already searched. The process continues until a defined point is found or the entire region with edges at DistFinal units from the reference point has been searched. If multiple defined points are found, the coordinates and value of the one closest to the reference point are returned. If no defined point is found within the region, an error is returned.

5.12. MpxGetEnd

Syntax:

MpxGetEnd(**FileNumber, QueryNumber**)

Parameters:

FileNumber—Number of a file opened by MpxInputFile, or zero
QueryNumber—Number of a query initiated by MpxGetStart, or zero

Function: This routine terminates one or more queries. If FileNumber is zero and QueryNumber is nonzero, the query specified by QueryNumber is terminated. If FileNumber is nonzero, all active queries to the file given by FileNumber are terminated. If both FileNumber and QueryNumber are zero, all active queries on all files are terminated. MpxGetEnd returns an error if FileNumber is nonzero and does not reference a file opened by MpxInputFile.

6. CONCLUSION

In developing the case study we have shown that the formalism described in Section 3.5 is useful for expressing functional GDP requirements, and we have identified several important technical issues involved in developing a production GDP requirements engineering environment. The most important issues are the need for a set of efficient primitives for reasoning about space and time, for high-volume data storage, and for graphic presentation of results. Examples of spatial primitives are operations which calculate the distance between two geographic objects and determine if one object is enclosed in another. It is essential that a system for simulating GDP requirements be capable of storing and processing very large amounts of informaton, and of reporting results both graphically and textually. The implementation language should be one which can concisely represent abstract data types and predicate calculus formulae. Prolog was a natural choice for an implementation language, although we found it necessary to enhance the language for high-volume data storage, and to provide graphic support separatly from the interpreter. Ideally, a GDP requirements engineering language should provide support for data storage, spatial reasoning, and graphics in a single package.

7. REFERENCES

[CLOC81] Clocksin, William F., and Mellish. Christopher S., *Programming in Prolog* Springer-Verlag, Berlin Heidelberg, 1981.

APPENDIX C

CSPS SYNTAX

1. BNF EXTENSIONS

- { } indicates that the enclosed sequence of symbols may be repeated zero or more times.

- [ ] indicates that the enclosed sequence of symbols is optional.

- Symbols that are in boldface are terminal symbols in the object language. **[**, **]** and **¦** (in the bold font) belong to CSPS; [,], and ¦ (in the normal font) belong to BNF.

- Portions of nonterminal symbol names that are in *italics* denote attributes, e.g., the type, of the non-terminal symbol name which follows the italicized portion.

2. SYSTEM-LEVEL SPECIFICATION

```
<csps-specification>     ::= <common-definitions> <function-definitions>
                              <process-definitions>

<common-definitions>     ::= <common-declaration> { ; <common-declaration> }

<common-declaration>     ::= <empty>
                           ¦  <type-declaration>
                           ¦  <constant-declaration>

<function-definitions>   ::= { <function> }

<process-definitions>    ::= <process> { <process> }
```

3. PROCESSES

```
<process>             ::=  <process-header> <block>

<process-header>      ::=  <community-tag> <label> { , <label> } ::

<community-tag>       ::=  MODULE ¦ SCHEDULE ¦ PROCESSOR ¦ ACTOR

<block>               ::=  [ <command-list> ]

<command-list>        ::=  { <local-declaration> ; ¦ <command> ; } <command>

<local-declaration>   ::=  <variable-declaration> ¦ <constant-declaration>
```

4. FUNCTIONS

<function> ::= <function-header> <block>

Note: The function <block> may not contain any i/o or synchronization commands. The name of the function is a variable with the same type as the function within the <block>; it's final value is returned as the value of the function.

<function-header> ::= <identifier> **(** <parameter-list> **)** **:** <type-spec> **::**

Note: In a <function-header>, the <type-spec> must not be an unconstrained array definition. However, it's <bound>s may be defined in terms of the values or sizes of the <parameter>s.

<parameter-list> ::= <empty> | <parameter> { **;** <parameter> }

<parameter> ::= <reg-variable-decl>

Note: The <parameter>s are implicitly constants within the body of the function. The <parameter>s may be unconstrained arrays; in this case, they take on the <bound>s of the actual arguments.

5. TYPE DECLARATIONS

<type-declaration> ::= **TYPE** <identifier> **IS** <type-definition>

<type-definition> ::= <type-name>
| <range>
| <enumeration-defn>
| <array-definition>
| <array-constraint>

<type-name> ::= <*type*-identifier>
| <builtin-type>
| <community-tag>

<builtin-type> ::= **INTEGER** | **BOOLEAN**
| **TYPE_ID** | **OBJECT_ID** | **FUNCTION**

<range> ::= <simple-expression>**..**<simple-expression>

Note: In <range>, both <simple-expression>'s may evaluate to any integer or user-defined enumeration values. However, the values must be of the same type and the value of the first expression must be ≤ the second.

<enumeration-defn> ::= **(** <identifier-list> **)**

Note: The above rule defines an ordered enumeration type. The enumeration values are listed in increasing order, i.e., if i is listed before j then i < j. MODULEs, SCHEDULEs, PROCESSORs, ACTORs, FUNCTIONs, TYPE_IDs, and OBJECT_IDs are implicitly defined enumeration types with undefined order.

<identifier-list> ::= <identfier> { **,** <identifier> }

6. ARRAY DECLARATIONS

<array-definition> ::= **ARRAY (**<bound-list> **) OF** <*scalar*-type-name>

Note: <*scalar*-type-name> must be a builtin type, community tag, or a user-defined enumeration or range type.

<array-constraint> ::= <*array*-type-name> **(** <bound-list> **)**

Note: If <*array*-type-name> is an array type defined with * <bound>'s, then each non-* <bound> in an <array-constraint> may constrain a * <bound> in the in the <array-definition> for <*array*-type-name>. The <bound>'s which have already been constrained are left blank. In object declarations, all <bound>'s must be contrained either by the object declaration or by a type definition.

<bound-list> ::= <bound> { **,** <bound> }

<bound> ::= <*positive-integer*-simple-expression>
| <range>
| <*enumeration*-type-name>
| **BOOLEAN**
| <empty>
| *

Note: An <*enumeration*-type-name> specified may be any enumeration type (either user-defined or implicit). An An <empty> bound may only be used in <array-constraint>'s for those <bound>'s which have been constrained by a previous TYPE definition.

7. OBJECT DECLARATIONS

```
<constant-declaration> ::= <identifier-list> : <type-spec> CONSTANT
                           [ := <value> ]

<variable-declaration> ::= <reg-variable-decl>
                         | <init-variable-decl>

<reg-variable-decl>    ::= <identifier-list> : <type-spec>

<init-variable-decl>   ::= <identifier-list> : <type-spec> := <value>

<type-spec>            ::= <type-name>
                         | <array-definition>
                         | <array-constraint>
                         | <range>

<value>                ::= <expression>
                         | ( <element-value> { , <element-value> } )

<element-value>        ::= <expression>
                         | <locator> : <expression>
```

Note: A <locator> specifies the position in the array that is being initialized by the <element-value>.

```
<locator>              ::= <expression>
                         | ( <expression-list> )
                         | OTHERS
```

8. COMMANDS

```
<command>              ::= <empty>
                         | <simple-command>
                         | <structured-command>

<simple-command>       ::= skip
                         | abort
                         | <assignment-command>
                         | <general-synch-command>
                         | <resynched-assignment>
                         | <resynched-i/o>

<assignment-command>   ::= <target> := <expression>

<target>               ::= <designator>
```

9. SYNCHRONIZATION COMMANDS

```
<general-synch-command>     ::= <composite-synch-command>
                              ¦ <integer-expression> * ( <composite-synch-command> )
                              ¦ <resynch-command>

<resynched-assignment>      ::= <resynch-expression> <assignment-command>

<resynched-i/o>             ::= <resynch-expression> <i/o-expression>
```

Note: <resynched-assignment>'s and <resynched-i/o>'s are translated into a sequence of three commands—a synchronization, then the assignment or i/o, and finally a resynchronization. If the command is in a guard, then the second and third commands are executed as the first two commands in the guarded <command-list>,

```
<resynch-command>           ::= <resynch-operator>

<composite-synch-command>   ::= <synch-command> { <synch-command> }

<synch-command>             ::= <synch-expression> ¦ <i/o-expression>

<synch-expression>          ::= <label> <synch-operator> <synch-pattern>

<resynch-expression>        ::= <label> <resynch-operator> <synch-pattern>

<i/o-expression>            ::= <process-label> <i/o-operator> <synch-pattern>

<synch-pattern>             ::= <empty> ¦ ( ) ¦ <synch-parameter>
                              ¦ ( <synch-parameter> { , <synch-parameter> } )

<synch-parameter>           ::= <expression> ¦ <designator> '
```

Note: Primes (') may not appear in <synch-pattern>'s following <i/o-operator>'s. The <expression>'s on input commands must be variable <identifier>'s. The <designator>'s which are followed by a prime must be either a variable name or array reference; function calls are not allowed.

```
<synch-operator>            ::= $ ¦ @ ¦ & ¦ %

<resynch-operator>          ::= $$ ¦ @@ ¦ && ¦ %%

<i/o-operator>              ::= ? ¦ !
```

10. STRUCTURED COMMANDS

```
<structured-command>           ::= <alternative-command>
                                 | <repetitive-command>
                                 | <guarded-selection-iteration>
                                 | <block>

<alternative-command>          ::= [ <guarded-command> { # <guarded-command> } ]

<repetitive-command>           ::= *<alternative-command>

<guarded-selection-iteration>  ::= +<alternative-command>

<guarded-command>              ::= <guard> → <command-list>

<guard>                        ::= <guard-list>
                                 | <composite-synch-command>
                                 | <guard-list> ; <composite-synch-command>

<guard-list>                   ::= <guard-element> { ; <guard-element> }

<guard-element>                ::= <boolean-expression>
                                 | <local-declaration>
```

11. EXPRESSIONS

```
<expression>                 ::= <simple-expression>
                                 [ <relational-operator> <simple-expression> ]

<relational-operator>        ::= = | ≠ | < | ≤ | > | ≥

<simple-expression>          ::= [ <unary-adding-operator> ] <term>
                                 { <addition-operator>  <term> }

<unary-adding-operator>      ::= + | −

<addition-operator>          ::= + | − | or

<term>                       ::= <factor> { <multiplication-operator> <factor> }

<multiplication-operator>    ::= * | / | and | mod | **

<factor>                     ::= <designator> | <integer> | <boolean>
                               | not <factor> | ( <expression> )
                               | <expansion-identifier> <expansion-modifier>
                               | ~  | ∞
```

Note: Above, <designator> may be a variable, constant, array reference, enumeration-literal, function call, or function name. The <expansion-identifier>'s may only appear in expressions appearing in <guard-element>'s.

```
<designator>                 ::= <identifier> [ ( <expression-list> ) ]

<expression-list>            ::= <expression> { , <expression> }
```

12. LABELS

```
<label>                 ::= <composite-identifier> <expansion-modifier>

<expansion-modifier>    ::= <empty>
                          ¦ ( <modifier-element> { ; <modifier-element> ) }

<modifier-element>      ::= <boolean-expression>
                          ¦ <simple-declaration>

<simple-declaration>    ::= <identifier-list> : <finite-scalar-type-name>
                          ¦ <identifier-list> : <range>

        Note:  The <finite-scalar-identifier> is a variable within the
               scope of the expansion.

<composite-identifier>  ::= <identifier> { . <expansion-element> }

<expansion-element>     ::= <identifier>
                          ¦ <expansion-identifier>
                          ¦ <unsigned>

<expansion-identifier>  ::= <variable-identifier>
```

13. LEXICAL ELEMENTS

```
<identifier> ::= <letter> { <letter> ¦ <digit> ¦ _ }

<letter>     ::= a ¦ b ¦ c ... ¦ y ¦ z ¦ A ¦ B ¦ C ... ¦ Y ¦ Z

<digit>      ::= 0 ¦ 1 ¦ 2 ... ¦ 8 ¦ 9

<integer>    ::= <unsigned> ¦ +<unsigned>
               ¦ −<unsigned>

<unsigned>   ::= <digit> { <digit> }

<boolean>    ::= true ¦ false

<comment>    ::= /* <general-text> */
```

Note: <comment>'s may appear anywhere a blank can appear, i.e., between any two lexical tokens. The above grammar assumes that comments are removed by a lexical analysis phase.

14. GENERIC BUILTIN FUNCTIONS

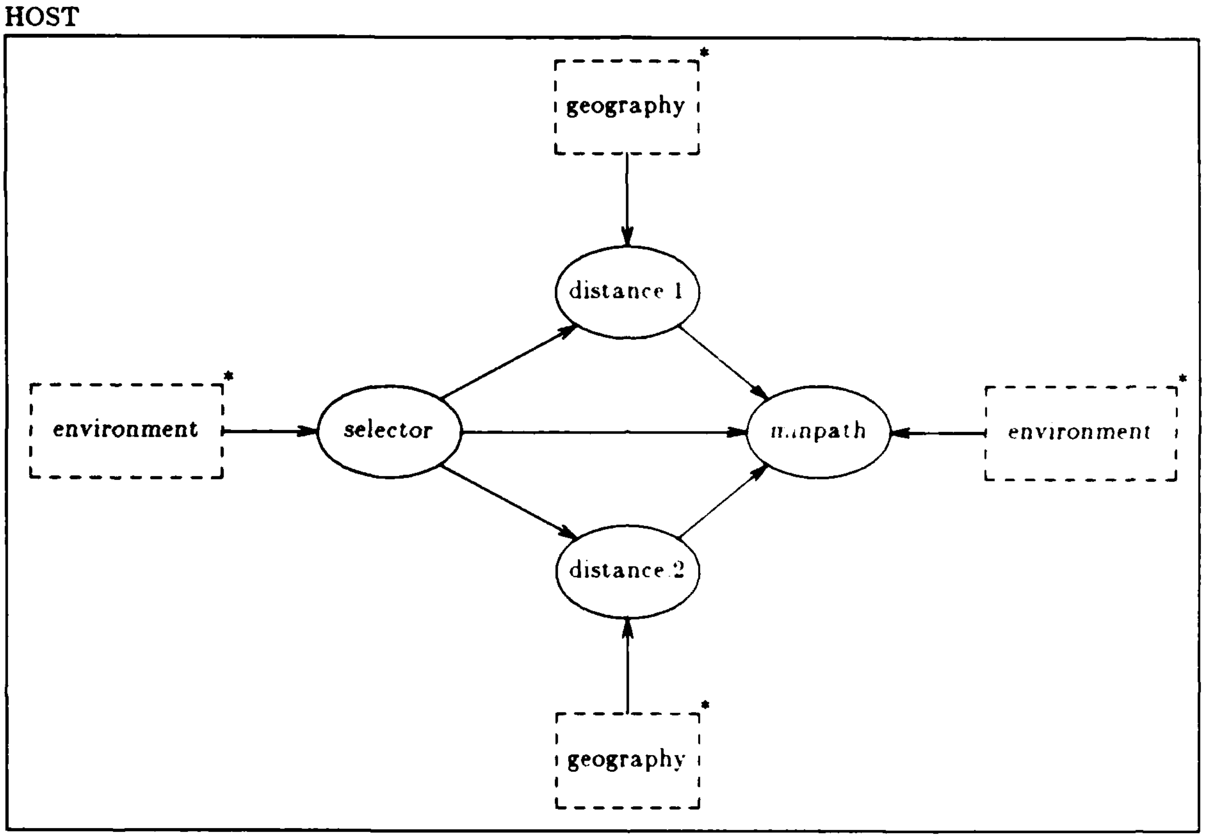
Explicit type conversion functions convert their arguments to the type:
INTEGER, BOOLEAN, and all user-defined types.

```
TYPE ( object: /* any type */ ) : TYPE_ID ::
        /* returns the internal type identifier for object */

ID ( object: /* any type */ ) : OBJECT_ID ::
        /* returns the internal identifier for object */

SIZE ( object: /* any type /* ) : INTEGER ::
        /* returns the size of object in units */

DIM ( object: /* any type */ ) : INTEGER ::
        /* returns the number of dimensions of array */

DIMSIZE ( object: /* any type */; dimension: INTEGER ) : INTEGER ::
        /* returns the number of elements of the given dimension */

LOBOUND ( object: /* any type */; dimension: INTEGER ) : /*the index type*/ ::
        /* returns the lower bound of the given dimension */

UPBOUND ( object: /* any type */; dimension: integer ) : /*the index type*/ ::
        /* returns the upper bound of the given dimension */

FIRST ( type: TYPE_ID ) : type ::
        /* returns the first value of the specified finite type */

LAST ( type: TYPE_ID ) : type ::
        /* returns the last value of the specified finite type */

SUCC ( value: /* any ordered scalar */ ) : /* same type as value */ ::
        /* returns the next greater value from type */

PRED ( value: /* any ordered scalar */ ) : /* same type as value */ ::
        /* returns the next lower value from type */
```

APPENDIX D

SAMPLE SYSTEM MODELS

STATIC ALLOCATION, UNIPROCESSOR, LOCAL COMMUNICATION



* This process is not modelled

1. SYSTEM FUNCTIONALITY

```
/*
Global Definitions - Definition of all variable types, global constants, and global functions.
*/


TYPE coordinate IS INTLGER;
TYPE point IS ARRAY(1..2) OF coordinate;
TYPE elevation IS INTEGER;
TYPE terrain IS ARRAY(*,*) OF elevation;
TYPE distance IS INTEGER;
TYPE relative_distance IS ARRAY(*,*) OF distance;
TYPE road_paths IS ARRAY(*,*) OF BOOLEAN;

x : INTEGER CONSTANT := 1;
y : INTEGER CONSTANT := 2;
e : INTEGER CONSTANT;

TYPE size IS INTEGER;
TYPE time IS INTEGER;

k : size CONSTANT;

TYPE host_service IS (allocation,deallocation,computation)

COST(s : host_service; q : size) : time ::
[
        [       s = allocation       →     COST := 1     #
                s = deallocation     →     COST := 1     #
                s = computation      →     COST := q     ]
];

COMPLEXITY(o : FUNCTION; q : size) : size ::
[
        [       o = Equ_distance     →     COMPLEXITY := q            #
                o = Min_distance     →     COMPLEXITY := 5 * q        ]
];
```

```
Area_spec_ok(e1,e2:point): BOOLEAN ::
[
        Area_spec_ok :=        e1(x) ≤ e2(x)  and  e1(y) ≤ e2(y)
];

End_points_ok(a1,a2,r1,r2:point): BOOLEAN ::
[

        End_points_ok :=       a1(x) ≥ r1(x) and a1(x) ≤ r2(x) and a1(y) ≥ r1(y) and a1(y) ≤ r2(y) and
                               a2(x) ≥ r1(x) and a2(x) ≤ r2(x) and a2(y) ≥ r1(y) and a2(y) ≤ r2(y)
];

Min_distance(D:relative_distance; T:terrain; e:integer): relative_distance ::
[
        /* body of function */
];

Equ_distance(D1,D2 : relative_distance; d : distance) : road_path ::
[
        /* body of function */
];


/*
        Specification of the environment and geography processes - The environment process
models the user interaction, getting the area boundary and road endpoints from user. The
geography is the terrain database from which elevations for a given area can be obtained.
*/


MODULE env1 ::
[
/* body of environment process */
]



MODULE geography ::
[
/* body of geography process */
]
```

```
/*
        The selector module gets the area boundary and road endpoints frome the environment.
and then checks that both endpoints of road are within the area given. If the endpoints are in
area, then the area boundary and road endpoints are sent to distance_1 and distance_2 to
determine the minimum distances to all points in area from both the first and second endpoints.
The area boundary is also sent to minpath, which will determine the points on the road.
*/


MODULE selector ::
[
f0 : MODULE CONSTANT := ~;

a1 : point; dcl.f0 $$ (SIZE(a1));
a2 : point; dcl.f0 $$ (SIZE(a2));
r1 : point; dcl.f0 $$ (SIZE(r1));
r2 : point; dcl.f0 $$ (SIZE(r2));

get.f0.env1 $$ (SIZE(a1)+SIZE(a2)+SIZE(r1)+SIZE(r2)) env1?(a1,a2,r1,r2);

[       Area_spec_ok(a1,a2)             → skip #
        not(Area_spec_ok(a1,a2))        → abort         ];

[       End_points_ok(a1,a2,r1,r2)      → skip #
        not(End_points_ok(a1,a2,r1,r2)) → abort         ];

send.f0.distance_1 $$ (SIZE(a1)+SIZE(a2)+SIZE(r1)+SIZE(r2)) distance_1!(a1,a2,r1,r2);
send.f0.distance_2 $$ (SIZE(a1)+SIZE(a2)+SIZE(r1)+SIZE(r2)) distance_2!(a1,a2,r1,r2);
send.f0.minpath $$ (SIZE(a1)+SIZE(a2)) minpath!(a1,a2);

dlt.f0 $$ (SIZE(a1)); dlt.f0 $$ (SIZE(a2));
dlt.f0 $$ (SIZE(r1)); dlt.f0 $$ (SIZE(r2))
]
```

```
/*
        Each distance module is given the boundary of an area and the endpoints of a road, and
then determines the minimum distance from the first road endpoint to every other point in the
area using the terrain information for the area that it gets from geography.
*/


MODULE distance_1 , distance_2 ::
[
f0 : MODULE CONSTANT := ~;

a1 : point; dcl.f0 $$ (SIZE(a1));
a2 : point; dcl.f0 $$ (SIZE(a2));
r1 : point; dcl.f0 $$ (SIZE(r1));
r2 : point; dcl.f0 $$ (SIZE(r2));
ok : BOOLEAN; dcl.f0 $$ (SIZE(ok));


get.f0.selector $$ (SIZE(a1)+SIZE(a2)+SIZE(r1)+SIZE(r2)) selector?(a1,a2,r1,r2);

[
        Z : terrain(a1(x)..a2(x),a1(y)..a2(y)); dcl.f0 $$ (SIZE(Z));
        D : relative_distance(a1(x)..a2(x),a1(y)..a2(y)); dcl.f0 $$ (SIZE(D));


        send.f0.geography $$ (SIZE(a1)+SIZE(a2)) geography!(a1,a2);
        get.f0.geography $$ (SIZE(ok)) geography?(ok);

        [       ok      → get.f0.geography $$ (SIZE(Z)) geography?(Z)  #
                not(ok) → abort                                         ];

        D(r1) := 0;
        eval.f0 $$ (Min_distance,SIZE(Z)) D := Min_distance(D,Z,e);

        [       D(r2)≠∞ → send.f0.minpath $$ (SIZE(D)+SIZE(r1)) minpath!(D,r1)      #
                D(r2)=∞ → abort                                                      ];

        dlt.f0 $$ (SIZE(Z)); dlt.f0 $$ (SIZE(D))
];

dlt.f0 $$ (SIZE(a1)); dlt.f0 $$ (SIZE(a2));
dlt.f0 $$ (SIZE(r1)); dlt.f0 $$ (SIZE(r2));
dlt.f0 $$ (SIZE(ok))
]
```

```
/*
        The minpath module gets the minimum distances of points in area from each road
endpoint, and determines the points that are on a road of minimum length between the
endpoints, and sends this list to the environment.
*/


MODULE minpath ::
[
f0 : MODULE CONSTANT := ~;

a1 : point; dcl.f0 $$ (SIZE(a1));
a2 : point; dcl.f0 $$ (SIZE(a2));
r1 : point; dcl.f0 $$ (SIZE(r1));
r2 : point; dcl.f0 $$ (SIZE(r2));

get.f0.selector $$ (SIZE(a1)+SIZE(a2)) selector?(a1,a2);

[
        D1 : relative_distance(a1(x)..a2(x),a1(y)..a2(y)); dcl.f0 $$ (SIZE(D1));
        D2 : relative_distance(a1(x)..a2(x),a1(y)..a2(y)); dcl.f0 $$ (SIZE(D2));

        +[      get.f0.distance_1 $$ (SIZE(D1),SIZE(r1)) distance_1?(D1,r1) → skip  #
                get.f0.distance_2 $$ (SIZE(D2),SIZE(r2)) distance_2?(D2,r2) → skip  ];

        [
                RD : road_paths(a1(x)..a2(x),a1(y)..a2(y)); dcl.f0 $$ (SIZE(RD));
                d : distance; dcl.f0 $$ (SIZE(d));

                d := D1(r2);
                eval.f0 $$ (Equ_distance,SIZE(D1)) RD := Equ_distance(D1,D2,d);
                        send.f0.env1 $$ (SIZE(RD)) env1!RD;

                        dlt.f0 $$ (SIZE(RD)); dlt.f0 $$ (SIZE(d))
                ];

                dlt.f0 $$ (SIZE(D2)); dlt.f0 $$ (SIZE(D1))
        ];

        dlt.f0 $$ (SIZE(a1)); dlt.f0 $$ (SIZE(a2));
        dlt.f0 $$ (SIZE(r1)); dlt.f0 $$ (SIZE(r2))
]
```

2. SYSTEM ARCHITECTURE

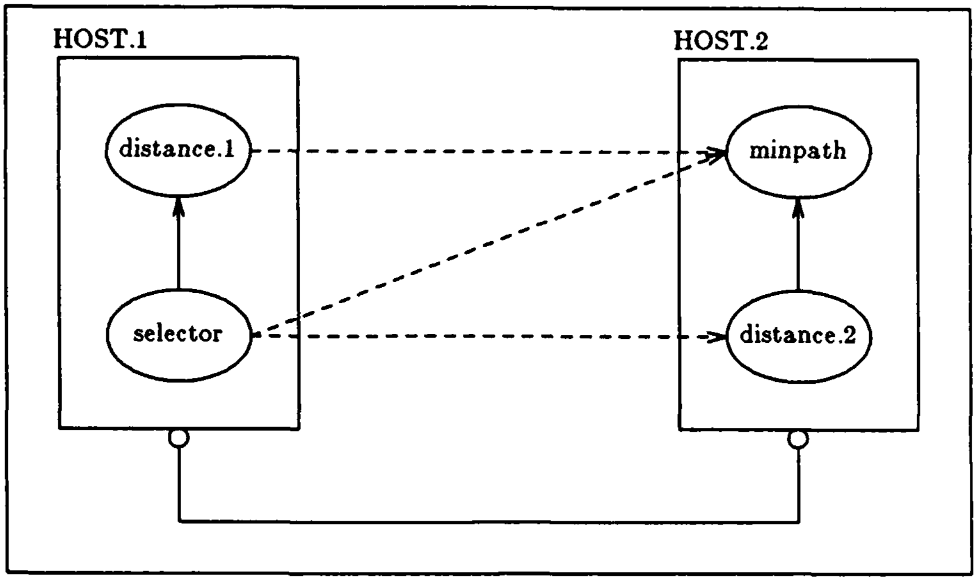
```
/*
            The host process models the operations required of our physical hardware. This
model is for memory allocation (total amount, not placement) so the operations modelled are
the allocation of memory and the deallocation (free) of memory.
*/


PROCESSOR host ::
[
        M : size := 0;
        n : size;
        p : MODULE;
        t : time;

        *[
                allocate.p" @ (n') →
                        [       M+n≤k → M := M+n; success.p @       #
                                M+n>k → skip; failure.p @           ];
                        t := COST(allocation,n); t*(tic &);
                        @@
                                                                            #
                free.p" @ (n') →
                        [       M-n≥0 → M := M−n; success.p @       #
                                M-n<0 → skip; failure.p @           ];
                        t := COST(deallocation,n); t*(tic &);
                        @@
                                                                            #
                compute.p" @ (n') →
                        t := COST(computation,n); t*(tic &);
                        @@
                                                                            #
                true → skip; tic &
        ]
]
```

```
/*
            Each schedule provides the interface between one of the modules and the host
process. In this model, this is limited to the allocation and deallocation of memory for variables
at the module level, and the allocation and deallocation of temporary storage used in the
transfer of data underlying the input/output commands.
*/


SCHEDULE server.f" (f : MODULE) ::
[
        n : size;
        o : FUNCTION;

        *[
                dcl.f $ (n')  → allocate.f @ (n);
                        [       success.f @ → skip;@@;$$        #
                                failure.f @ → abort             ]               #

                dlt.f $ (n') → free.f @ (n);
                        [       success.f @ → skip;@@;$$        #
                                failure.f @ → abort             ]               #

                eval.f $ (o',n') →
                        compute.f @ (COMPLEXITY(o,n));
                        @@;$$
                                                                                #

                send.f.p" $ (n') →
                        allocate.f @ (n);
                        [       success.f @ → skip;@@           #
                                failure.f @ → abort             ];
                        ready.f.p $ (n);
                        $$
                                                                                #
                get.f.p" $ (n') ready.p.f $ (n) →
                        free.f @ (n);
                        [       success.f @ → skip;@@           #
                                failure.f @ → abort             ]
                $$
        ]
]
```

3. PERFORMANCE SPECIFICATION

```
/*
        The actor clock models a global clock that sychronizes with all hosts on every tick of
the clock.
*/


ACTOR clock ::
[
        T : time := 0;

        *[      true → tic &; T := T+1;         ]
]
```

STATIC ALLOCATION, DUAL-PROCESSORS, POINT TO POINT COMMUNICATION



LEGEND

———	Physical Communication Links
- - - - -	Logical Communication Links
○	Ports

1. SYSTEM FUNCTIONALITY

```
/*
Global Definitions - Definition of all variable types, global constants, and global functions.
*/

TYPE coordinate IS INTEGER;
TYPE point IS ARRAY(1..2) OF coordinate;
TYPE elevation IS INTEGER;
TYPE terrain IS ARRAY(*,*) OF elevation;
TYPE distance IS INTEGER;
TYPE relative_distance IS ARRAY(*,*) OF distance;
TYPE road_paths IS ARRAY(*,*) OF BOOLEAN;

x : INTEGER CONSTANT := 1;
y : INTEGER CONSTANT := 2;
e : INTEGER CONSTANT ;

TYPE size IS INTEGER;
TYPE time IS INTEGER;

k : size CONSTANT;

TYPE host_service IS (allocation,deallocation,computation);
TYPE processing_element IS (1,2);
TYPE module_to_processor_mapping IS ARRAY(MODULE) OF PROCESSOR;

HOST_OF : module_to_processor_mapping CONSTANT :=
        (selector : host.1; distance_1 : host.1; distance_2 : host.2;
         minpath : host.2; envl : host.1; geography : host.2);

COST(s : host_service; q : size) : time ::
[
        [       s = allocation          →       COST := 1       #
                s = deallocation        →       COST := 1       #
                s = computation         →       COST := q       ]
];

PCOST(h1,h2 : PROCESSOR; q : size) : time ::
[
        [       (h1 = host.1 and h2 = host.2    →       PCOST := q      #
                (h1 = host.2 and h2 = host.1    →       PCOST := q      ]
];

COMPLEXITY(o : FUNCTION; q : size) : size ::
[
        [       o = Equ_distance        →       COMPLEXITY := q                 #
                o = Min_distance        →       COMPLEXITY := 5 * q             ]
];
```

```
Area_spec_ok(e1,e2:point): BOOLEAN ::
[
        Area_spec_ok :=         e1(x) ≤ e2(x)  and  e1(y) ≤ e2(y)
];

End_points_ok(a1,a2,r1,r2:point): BOOLEAN ::
[

        End_points_ok :=        a1(x) ≥ r1(x) and a1(x) ≤ r2(x) and a1(y) ≥ r1(y) and a1(y) ≤ r2(y) and
                                a2(x) ≥ r1(x) and a2(x) ≤ r2(x) and a2(y) ≥ r1(y) and a2(y) ≤ r2(y)
];

Min_distance(D:relative_distance; T:terrain; e:integer): relative_distance ::
[
        /* body of function */
];

Equ_distance(D1,D2 : relative_distance; d : distance) : road_path ::
[
        /* body of function */
];


/*
        Specification of the environment and geography processes - The environment process
models the user interaction, getting the area boundary and road endpoints from user. The
geography is the terrain database from which elevations for a given area can be obtained.
*/


MODULE env1 ::
[
/* body of environment process */
]



MODULE geography ::
[
/* body of geography process */
]
```

```
/*
        The selector module gets the area boundary and road endpoints frome the environment.
and then checks that both endpoints of road are within the area given. If the endpoints are in
area, then the area boundary and road endpoints are sent to distance_1 and distance_2 to
determine the minimum distances to all points in area from both the first and second endpoints.
The area boundary is also sent to minpath, which will determine the points on the road.
*/


MODULE selector ::
[
f0 : MODULE CONSTANT := ~;

a1 : point; dcl.f0 $$ (SIZE(a1));
a2 : point; dcl.f0 $$ (SIZE(a2));
r1 : point; dcl.f0 $$ (SIZE(r1));
r2 : point; dcl.f0 $$ (SIZE(r2));

get.f0.env1 $$ (SIZE(a1)+SIZE(a2)+SIZE(r1)+SIZE(r2)) env1?(a1,a2,r1,r2);

[       Area_spec_ok(a1,a2)             → skip #
        not(Area_spec_ok(a1,a2))        → abort         ];

[       End_points_ok(a1,a2,r1,r2)      → skip #
        not(End_points_ok(a1,a2,r1,r2)) → abort         ];

send.f0.distance_1 $$ (SIZE(a1)+SIZE(a2)+SIZE(r1)+SIZE(r2)) distance_1!(a1,a2,r1,r2);
send.f0.distance_2 $$ (SIZE(a1)+SIZE(a2)+SIZE(r1)+SIZE(r2)) distance_2!(a1,a2,r1,r2);
send.f0.minpath $$ (SIZE(a1)+SIZE(a2)) minpath!(a1,a2);

dlt.f0 $$ (SIZE(a1)); dlt.f0 $$ (SIZE(a2));
dlt.f0 $$ (SIZE(r1)); dlt.f0 $$ (SIZE(r2))
]
```

```
/*
        Each distance module is given the boundary of an area and the endpoints of a road, and
then determines the minimum distance from the first road endpoint to every other point in the
area using the terrain information for the area that it gets from geography.
*/


MODULE distance_1 , distance_2 ::
[
f0 : MODULE CONSTANT := ˜;

a1 : point; dcl.f0 $$ (SIZE(a1));
a2 : point; dcl.f0 $$ (SIZE(a2));
r1 : point; dcl.f0 $$ (SIZE(r1));
r2 : point; dcl.f0 $$ (SIZE(r2));
ok : BOOLEAN; dcl.f0 $$ (SIZE(ok));


get.f0.selector $$ (SIZE(a1)+SIZE(a2)+SIZE(r1)+SIZE(r2)) selector?(a1,a2,r1,r2);

[
        Z : terrain(a1(x)..a2(x),a1(y)..a2(y)); dcl.f0 $$ (SIZE(Z));
        D : relative_distance(a1(x)..a2(x),a1(y)..a2(y)); dcl.f0 $$ (SIZE(D));


        send.f0.geography $$ (SIZE(a1)+SIZE(a2)) geography!(a1,a2);
        get.f0.geography $$ (SIZE(ok)) geography?(ok);

        [       ok      → get.f0.geography $$ (SIZE(Z)) geography?(Z)  #
                not(ok) → abort                                          ];

        D(r1) := 0;
        eval.f0 $$ (Min_distance,SIZE(Z)) D := Min_distance(D,Z,e);

        [       D(r2)≠∞ → send.f0.minpath $$ (SIZE(D)+SIZE(r1)) minpath!(D,r1)      #
                D(r2)=∞ → abort                                                      ];

        dlt.f0 $$ (SIZE(Z)); dlt.f0 $$ (SIZE(D))
];

dlt.f0 $$ (SIZE(a1)); dlt.f0 $$ (SIZE(a2));
dlt.f0 $$ (SIZE(r1)); dlt.f0 $$ (SIZE(r2));
dlt.f0 $$ (SIZE(ok))
]
```

```
/*
        The minpath module gets the minimum distances of points in area from each road
endpoint, and determines the points that are on a road of minimum length between the
endpoints, and sends this list to the environment.
*/


MODULE minpath ::
[
f0 : MODULE CONSTANT := ~;

a1 : point; dcl.f0 $$ (SIZE(a1));
a2 : point; dcl.f0 $$ (SIZE(a2));
r1 : point; dcl.f0 $$ (SIZE(r1));
r2 : point; dcl.f0 $$ (SIZE(r2));

get.f0.selector $$ (SIZE(a1)+SIZE(a2)) selector?(a1,a2);

[
        D1 : relative_distance(a1(x)..a2(x),a1(y)..a2(y)); dcl.f0 $$ (SIZE(D1));
        D2 : relative_distance(a1(x)..a2(x),a1(y)..a2(y)); dcl.f0 $$ (SIZE(D2));

        +[      get.f0.distance_1 $$ (SIZE(D1),SIZE(r1)) distance_1?(D1,r1) → skip  #
                get.f0.distance_2 $$ (SIZE(D2),SIZE(r2)) distance_2?(D2,r2) → skip  ];

        [
                RD : road_paths(a1(x)..a2(x),a1(y)..a2(y)); dcl.f0 $$ (SIZE(RD));
                d : distance; dcl.f0 $$ (SIZE(d));

                d := D1(r2);
                eval.f0 $$ (Equ_distance,SIZE(D1)) RD := Equ_distance(D1,D2,d);
                        send.f0.env1 $$ (SIZE(RD)) env1!RD;

                        dlt.f0 $$ (SIZE(RD)); dlt.f0 $$ (SIZE(d))
                ];

                dlt.f0 $$ (SIZE(D2)); dlt.f0 $$ (SIZE(D1))
        ];

        dlt.f0 $$ (SIZE(a1)); dlt.f0 $$ (SIZE(a2));
        dlt.f0 $$ (SIZE(r1)); dlt.f0 $$ (SIZE(r2))
]
```

2. SYSTEM ARCHITECTURE

```
/*
        The host process models the operations required of our physical hardware. This model is
for memory allocation (total amount, not placement) so the operations modelled are the
allocation of memory and the deallocation (free) of memory.
*/


PROCESSOR host.i" (i : processing_element) ::
[
        M : size := 0;
        n : size;
        p,q : MODULE;
        t : time;
        h : PROCESSOR;
        h0 : PROCESSOR CONSTANT := ~ ;

        *[
                allocate.h0.p" @ (n') →
                        [       M+n≤k → M := M+n; success.h0.p @   #
                                M+n>k → skip; failure.h0.p @         ];
                        t := COST(allocation,n); t*(tic &); @@
                                                                        #
                free.h0.p" @ (n') →
                        [       M-n≥0 → M := M−n; success.h0.p @   #
                                M-n<0 → skip; failure.h0.p @         ];
                        t := COST(deallocation,n); t*(tic &); @@
                                                                        #
                compute.h0.p" @ (n') →
                        t := COST(computation,n); t*(tic &); @@
                                                                        #
                ready.h0.h".p".q" @ (n') h!(n) →
                        skip; @@
                                                                        #
                ready.h".h0.p".q" @ (n') h?(n) →
                        skip; @@
                                                                        #
                pass.h0.h".p".q" @ (n') h!(n) →
                        t := PCOST(h0,h,n); t*(tic &); @@
                                                                        #
                pass.h".h0.p".q" @ (n') h?(n) →
                        t := PCOST(h,h0,n); t*(tic &); @@
                                                                        #
                true → skip: tic &
        ];
```

3. SYSTEM SCHEDULER

/*

Each schedule provides the interface between one of the modules and the host process. In this model, this is limited to the allocation and deallocation of memory for variables at the module level, and the allocation and deallocation of temporary storage used in the transfer of data underlying the input/output commands.

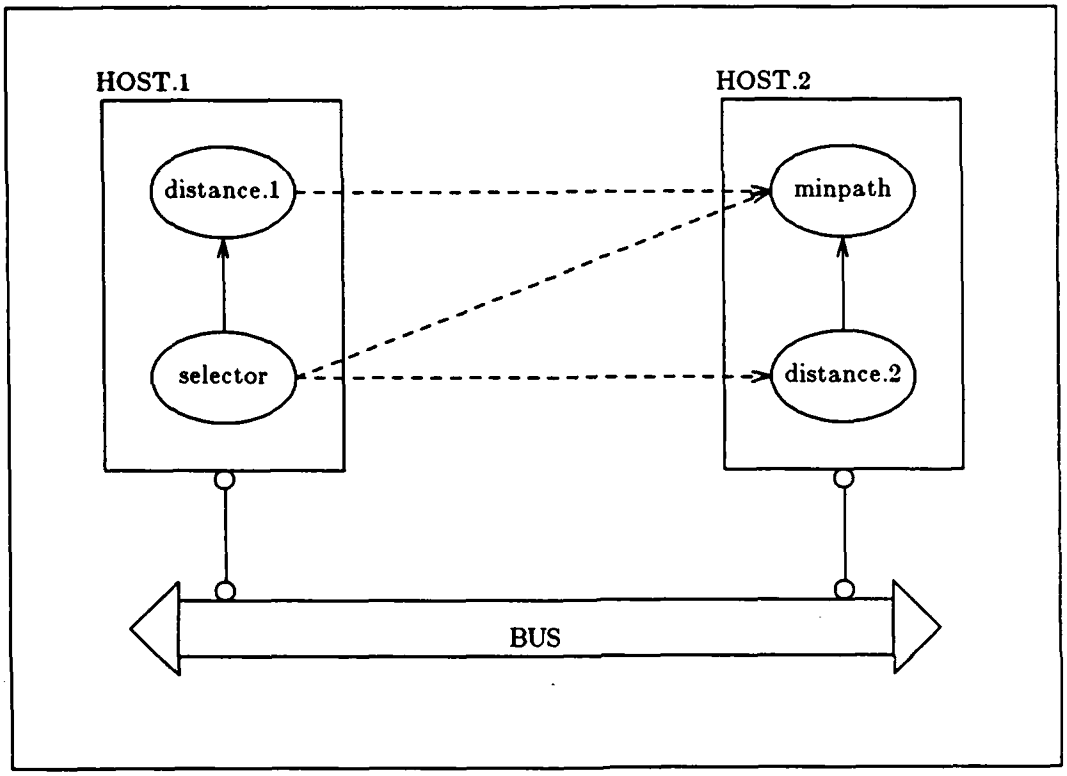
*/

```
SCHEDULE server.f" (f : MODULE) ::
[
        n : size;
        o : MODULE;
        p : MODULE;
        h : PROCESSOR;
        h0 : PROCESSOR CONSTANT := HOST_OF(˜);

        *[
                dcl.f $ (n') → allocate.h0.f @ (n);
                        [       success.h0.f @ → skip;@@;$$   #
                                failure.h0.f @ → abort        ]                #

                dlt.f $ (n') → free.h0.f @ (n);
                        [       success.h0.f @ → skip;@@;$$   #
                                failure.h0.f @ → abort        ]                #

                eval.f $ (o',n') →
                        compute.h0.f @ (COMPLEXITY(o,n));
                        @@;$$
                                                                               #

                send.f.p" $ (n') →
                        h := HOST_OF(p);
                        [       h ≠ h0 →
                                        allocate.h0.f @ (n);
                                        [       success.h0.f @ → skip;@@      #
                                                failure.h0.f @ → abort        ];
                                        ready.h0.h.f.p @ (n); @@;
                                        pass.h0.h.f.p @ (n); @@;
                                        free.h0.f @ (n);
                                        [       success.h0.f @ → skip;@@      #
                                                failure.h0.f @ → abort        ];
                                        $$
                                h = h0 →
                                        allocate.h0.f @ (n);
                                        [       success.h0.f @ → skip;@@      #
                                                failure.h0.f @ → abort        ];
                                        ready.h0.f.p $ (n);
                                        $$
                                                                               ]
                                                                                   #
get.f.p" $ (n') ready.h0.p.f @ (n) →
          skip; @@;
          free.h0.f @ (n);
          [         success.h0.f @ → skip;@@        #
                    failure.h0.f @ → abort ];
          $$
                                                                        #
get.f.p" $ (n') ready.HOST_OF(p).h0.p.f $ (n) →
          allocate.h0.f @ (n);
          [         success.h0.f @ → skip;@@        #
                    failure.h0.f @ → abort ]
          pass.HOST_OF(p).h0.p.f @ (n);@@;
          free.h0.f @ (n);
          [         success.h0.f @ → skip;@@        #
                    failure.h0.f @ → abort ]
          $$
                                                                   ]
       ]
  ]
```

4. PERFORMANCE SPECIFICATION

```
/*
        The actor clock models a global clock that sychronizes with all hosts on every tick of
the clock.
*/



ACTOR clock ::
[
        T : time := 0;

        *[        true → tic &; T := T+1;          ]
]
```

STATIC ALLOCATION, DUAL-PROCESSORS, BUS COMMUNICATION



LEGEND

———	Physical Communication Links
- - - - -	Logical Communication Links
○	Ports

1. SYSTEM FUNCTIONALITY

```
/*
Global Definitions - Definition of all variable types, global constants, and global functions.
*/


TYPE coordinate IS INTEGER;
TYPE point IS ARRAY(1..2) OF coordinate;
TYPE elevation IS INTEGER;
TYPE terrain IS ARRAY(*,*) OF elevation;
TYPE distance IS INTEGER;
TYPE relative_distance IS ARRAY(*,*) OF distance;
TYPE road_paths IS ARRAY(*,*) OF BOOLEAN;

x : INTEGER CONSTANT := 1;
y : INTEGER CONSTANT := 2;
e : INTEGER CONSTANT ;

TYPE size IS INTEGER;
TYPE time IS INTEGER;

k : size CONSTANT;

TYPE processing_element IS (1,2);
TYPE bus_unit IS (1);
TYPE host_service IS (allocation,deallocation,computation);
TYPE module_to_processor_mapping IS ARRAY(MODULE) OF PROCESSOR;

TYPE packet IS ARRAY(1..3) of INTEGER;

toprocess : INTEGER CONSTANT := 1;
fromprocess : INTEGER CONSTANT := 2;
packetsize : INTEGER CONSTANT := 3;
ACKNOWLEDGEMENT : INTEGER CONSTANT := -1;

HOST_OF : module_to_processor_mapping CONSTANT :=
        (selector : host.1; distance_1 : host.1; distance_2 : host.2;
         minpath : host.2; env1 : host.1; geography : host.2);

SCHEDULE_OF(m : MODULE) : SCHEDULE ::
[
        SCHEDULE_OF := server.m
];
```

```
is_real_processor(p : PROCESSOR) : BOOLEAN ::
[
        pe : processing_element;
        be : bus_unit;
        [       p = host.pe"    →       is_real_processor := true       #
                p = bus.be"     →       is_real_processor := false      ]
];

COST(s : host_service; q : size) : time ::
[
        [       s = allocation          →       COST := 1       #
                s = deallocation        →       COST := 1       #
                s = computation         →       COST := q       ]
];

PCOST(h1,h2 : PROCESSOR; q : size) : time ::
[
        [       (h1 = host.1 and h2 = host.2)   →       PCOST := q      #
                (h1 = host.2 and h2 = host.1)   →       PCOST := q      ]
];

COMPLEXITY(o : FUNCTION ; q : size) : size ::
[
        [       o = Equ_distance        →       COMPLEXITY := q         #
                o = Min_distance        →       COMPLEXITY := 5 * q     ]
];
allocated_to (f : MODULE; h : PROCESSOR) : BOOLEAN ::
[
        allocated_to := (h = host.1 and f = selector) or (h = host.1 and f = distance_1) or
                        (h = host.2 and f = distance_2) or (h = host.2 and f = minpath)
];

connected (h : PROCESSOR; b : PROCESSOR) : BOOLEAN ::
[
        connected := (h = host.1 and b = bus.1) or (h = host.2 and b = bus.1)
];
```

```
Area_spec_ok(e1,e2:point): BOOLEAN ::
[
        Area_spec_ok :=         e1(x) ≤ e2(x)  and  e1(y) ≤ e2(y)
];

End_points_ok(a1,a2,r1,r2:point): BOOLEAN ::
[

        End_points_ok :=        a1(x) ≥ r1(x) and a1(x) ≤ r2(x) and a1(y) ≥ r1(y) and a1(y) ≤ r2(y) and
                                a2(x) ≥ r1(x) and a2(x) ≤ r2(x) and a2(y) ≥ r1(y) and a2(y) ≤ r2(y)
];

Min_distance(D:relative_distance; T:terrain; e:integer): relative_distance ::
[
        /* body of function */
];

Equ_distance(D1,D2 : relative_distance; d : distance) : road_path ::
[
        /* body of function */
];


/*
        Specification of the environment and geography processes - The environment process
models the user interaction, getting the area boundary and road endpoints from user. The
geography is the terrain database from which elevations for a given area can be obtained.
*/


MODULE env1 ::
[
/* body of environment process */
]



MODULE geography ::
[
/* body of geography process */
]
```

```
/*
        The selector module gets the area boundary and road endpoints frome the environment,
and then checks that both endpoints of road are within the area given. If the endpoints are in
area, then the area boundary and road endpoints are sent to distance_1 and distance_2 to
determine the minimum distances to all points in area from both the first and second endpoints.
The area boundary is also sent to minpath, which will determine the points on the road.
*/


MODULE selector ::
[
f0 : MODULE CONSTANT := ~;

a1 : point; dcl.f0 $$ (SIZE(a1));
a2 : point; dcl.f0 $$ (SIZE(a2));
r1 : point; dcl.f0 $$ (SIZE(r1));
r2 : point; dcl.f0 $$ (SIZE(r2));

get.f0.env1 $$ (SIZE(a1)+SIZE(a2)+SIZE(r1)+SIZE(r2)) env1?(a1,a2,r1,r2);

[       Area_spec_ok(a1,a2)              → skip #
        not(Area_spec_ok(a1,a2))         → abort      ];

[       End_points_ok(a1,a2,r1,r2)       → skip #
        not(End_points_ok(a1,a2,r1,r2))  → abort      ];

send.f0.distance_1 $$ (SIZE(a1)+SIZE(a2)+SIZE(r1)+SIZE(r2)) distance_1!(a1,a2,r1,r2);
send.f0.distance_2 $$ (SIZE(a1)+SIZE(a2)+SIZE(r1)+SIZE(r2)) distance_2!(a1,a2,r1,r2);
send.f0.minpath $$ (SIZE(a1)+SIZE(a2)) minpath!(a1,a2);

dlt.f0 $$ (SIZE(a1)); dlt.f0 $$ (SIZE(a2));
dlt.f0 $$ (SIZE(r1)); dlt.f0 $$ (SIZE(r2))
]
```

```
/*
        Each distance module is given the boundary of an area and the endpoints of a road, and
then determines the minimum distance from the first road endpoint to every other point in the
area using the terrain information for the area that it gets from geography.
*/


MODULE distance_1 , distance_2 ::
[
f0 : MODULE CONSTANT := ~;

a1 : point; dcl.f0 $$ (SIZE(a1));
a2 : point; dcl.f0 $$ (SIZE(a2));
r1 : point; dcl.f0 $$ (SIZE(r1));
r2 : point; dcl.f0 $$ (SIZE(r2));
ok : BOOLEAN; dcl.f0 $$ (SIZE(ok));


get.f0.selector $$ (SIZE(a1)+SIZE(a2)+SIZE(r1)+SIZE(r2)) selector?(a1,a2,r1,r2);

[
        Z : terrain(a1(x)..a2(x),a1(y)..a2(y)); dcl.f0 $$ (SIZE(Z));
        D : relative_distance(a1(x)..a2(x),a1(y)..a2(y)); dcl.f0 $$ (SIZE(D));


        send.f0.geography $$ (SIZE(a1)+SIZE(a2)) geography!(a1,a2);
        get.f0.geography $$ (SIZE(ok)) geography?(ok);

        [       ok      → get.f0.geography $$ (SIZE(Z)) geography?(Z)  #
                not(ok) → abort                                          ];

        D(r1) := 0;
        eval.f0 $$ (Min_distance,SIZE(Z)) D := Min_distance(D,Z,e);

        [       D(r2)≠∞ → send.f0.minpath $$ (SIZE(D)+SIZE(r1)) minpath!(D,r1)      #
                D(r2)=∞ → abort                                                      ];

        dlt.f0 $$ (SIZE(Z)); dlt.f0 $$ (SIZE(D))
];

dlt.f0 $$ (SIZE(a1)); dlt.f0 $$ (SIZE(a2));
dlt.f0 $$ (SIZE(r1)); dlt.f0 $$ (SIZE(r2));
dlt.f0 $$ (SIZE(ok))
]
```

```
/*
        The minpath module gets the minimum distances of points in area from each road
endpoint, and determines the points that are on a road of minimum length between the
endpoints, and sends this list to the environment.
*/


MODULE minpath ::
[
f0 : MODULE CONSTANT := ˜;

a1 : point; dcl.f0 $$ (SIZE(a1));
a2 : point; dcl.f0 $$ (SIZE(a2));
r1 : point; dcl.f0 $$ (SIZE(r1));
r2 : point; dcl.f0 $$ (SIZE(r2));

get.f0.selector $$ (SIZE(a1)+SIZE(a2)) selector?(a1,a2);

[
        D1 : relative_distance(a1(x)..a2(x),a1(y)..a2(y)); dcl.f0 $$ (SIZE(D1));
        D2 : relative_distance(a1(x)..a2(x),a1(y)..a2(y)); dcl.f0 $$ (SIZE(D2));

        +[      get.f0.distance_1 $$ (SIZE(D1),SIZE(r1)) distance_1?(D1,r1) → skip  #
                get.f0.distance_2 $$ (SIZE(D2),SIZE(r2)) distance_2?(D2,r2) → skip  ];

        [
                RD : road_paths(a1(x)..a2(x),a1(y)..a2(y)); dcl.f0 $$ (SIZE(RD));
                d : distance; dcl.f0 $$ (SIZE(d));

                d := D1(r2);
                eval.f0 $$ (Equ_distance,SIZE(D1)) RD := Equ_distance(D1,D2,d);
                        send.f0.env1 $$ (SIZE(RD)) env1!RD;

                        dlt.f0 $$ (SIZE(RD)); dlt.f0 $$ (SIZE(d))
                ];

                dlt.f0 $$ (SIZE(D2)); dlt.f0 $$ (SIZE(D1))
        ];

        dlt.f0 $$ (SIZE(a1)); dlt.f0 $$ (SIZE(a2));
        dlt.f0 $$ (SIZE(r1)); dlt.f0 $$ (SIZE(r2))
]
```

2. SYSTEM ARCHITECTURE

/*

The host process models the operations required of our physical hardware. This model is for memory allocation (total amount, not placement) so the operations modelled are the allocation of memory and the deallocation (free) of memory.

*/

```
PROCESSOR host.i" (i : processing_element) ::
[
        M : size := 0;
        n : size;
        p : MODULE;
        t : time;
        h : PROCESSOR;
        h0 : PROCESSOR CONSTANT := ˜ ;

        *[
                allocate.h0.p" @ (n') →
                        [       M+n≤k → M := M+n; success.h0.p @    #
                                M+n>k → skip; failure.h0.p @         ];
                        t := COST(i,allocation,n); t*(tic &);
                        @@
                                                                          #
                free.h0.p" @ (n') →
                        [       M-n≥0 → M := M−n; success.h0.p @    #
                                M-n<0 → skip; failure.h0.p @         ];
                        t := COST(i,deallocation,n); t*(tic &);
                        @@
                                                                          #
                compute.h0.p" @ (n') →
                        t := COST(i,computation,n); t*(tic &);
                        @@
                                                                          #
                transfer.h0.h" @ (x') bus.1!(h,x) →
                        t := PCOST(bus.1,x); t*(tic &);
                        @@
                                                                          #

                bus.1?(x) receive.h0.h" @ (x) →
                        t := PCOST(bus.1,x); t*(tic &);
                        @@
                                                                          #

                true → tic & skip
        ]
]
```

```
PROCESSOR bus.i" (i : bus_unit) ::
[
        b : BOOLEAN := true;
        x : PACKET;
        h0 : PROCESSOR CONSTANT := ~. ;
        h1,h2 : PROCESSOR;

        *[
                connected(h1",h0); h1?(h2,x) →
                        *[ b; h2!(x) → b := false;]
                        b := true;
        ]
]
```

3. SYSTEM SCHEDULER

/*

Each schedule provides the interface between one of the modules and the host process. In this model, this is limited to the allocation and deallocation of memory for variables at the module level, and the allocation and deallocation of temporary storage used in the transfer of data underlying the input/output commands.

*/

```
SCHEDULE server.f" (f : MODULE) ::
[
        n : INTEGER;
        o : MODULE; p : MODULE;
        h : PROCESSOR;
        h0 : PROCESSOR CONSTANT := HOST_OF(˜);

        *[
                dcl.f $ (n') → allocate.h0.f @ (n);
                        [       success.h0.f @ → skip;@@;$$   #
                                failure.h0.f @ → abort        ]                #

                dlt.f $ (n') → free.h0.f @ (n);
                        [       success.h0.f @ → skip;@@;$$   #
                                failure.h0.f @ → abort        ]                #

                eval.f $ (o',n') →
                        compute.h0.f @ (COMPLEXITY(o,n));
                        @@;$$
                                                                               #

                send.f.p" $ (n') →
                        h := HOST_OF(p);
                        [       h ≠ h0 →
                                        allocate.h0.f @ (n);
                                        [       success.h0.f @ → skip;@@      #
                                                failure.h0.f @ → abort        ];
                                        bus_daemon.h0!(h,f,p,n,SENDIT);
                                        bus_daemon.h0?(ack);
                                        free.h0.f @ (n);
                                        [       success.h0.f @ → skip;@@      #
                                                failure.h0.f @ → abort        ];
                                        $$
                                h = h0 →
                                        allocate.h0.f @ (n);
                                        [       success.h0.f @ → skip;@@      #
                                                failure.h0.f @ → abort        ];
                                        ready.h0.f.p $;
                                        $$

                                                                               #
                get.f.p" $ (n') ready.h0.p.f @ (n) →
                        free.h0.f @ (n);
                        [           success.h0.f @ → skip;@@        #
                                    failure.h0.f @ → abort  ];
                        $$
                                                                                          #
                get.f.p" $ (n') have_data.h".h0.p.f (h = HOST_OF(p)) $ (n) →
                        allocate.h0.f @ (n);
                        [           success.h0.f @ → skip;@@        #
                                    failure.h0.f @ → abort  ];
                        bus_daemon.h0?();
                        free.h0.f @ (n);
                        [           success.h0.f @ → skip;@@        #
                                    failure.h0.f @ → abort  ];
                        $$
          ]
  ]
```

```
/*
        The Daemon runs the bus.
*/


SCHEDULE bus_daemon.i" (i : PROCESSOR; is_real_processor(i)) ::
[
        s : SCHEDULE;
        pf,pt : MODULE;  f : MODULE;
        h : PROCESSOR;  h0 : PROCESSOR CONSTANT := i;
        n : size;
        x : PACKET;
        dir : INTEGER;
        w,id : ARRAY(MODULE,MODULE) of INTEGER;

        *[
                (allocated_to(f",h0); server.f?(h,pf,pt,n,dir) →
                        [       dir = SENDIT →
                                        x := (pf,pt,n);
                                        transfer.h0.h @ (x); @@
                                                                        #
                                dir = GETIT →
                                        w(pf,pt) := n;
                        ]
                                                                                #
                receive.h0.h" @ (x) →
                        pf := x(fromprocess);
                        pt := x(toprocess);
                        n := x(packetsize);
                        [       n =  ACKNOWLEDGEMENT  →
                                        s := SCHEDULE_OF(pf);
                                        s!();
                                                                        #
                                n ≠ ACKNOWLEDGEMENT →
                                        id(pf,pt) := n;
                        ]
                        skip; @@
                                                                                #

                id(pf",pt") > 0; w(pf,pt) > 0; have_data.h".h0.pf.pt (h = HOST_OF(pf)) $ →
                        s := SCHEDULE_OF(pt); s!();
                        id(pf,pt) := 0;
                        x(fromprocess) := pf;
                        x(toprocess) := pt;
                        x(packetsize) := ACKNOWLEDGEMENT;
                        transfer.h0.h) @ (x); @@
        ]
]
```

4. PERFORMANCE SPECIFICATION

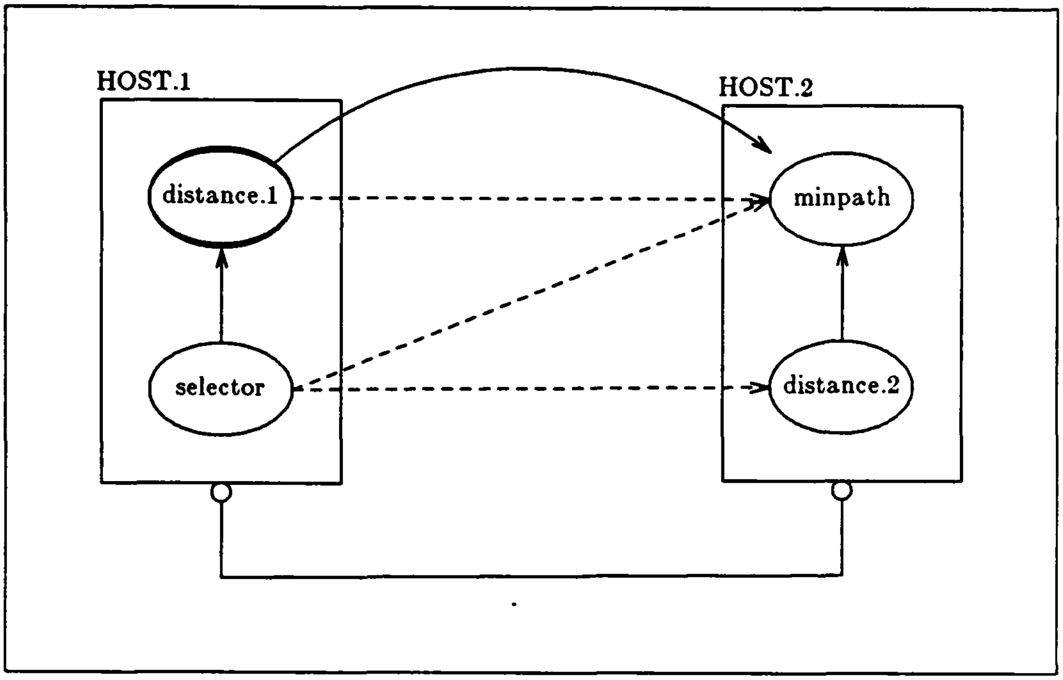
/*

The actor clock mod^els a global clock that sychronizes with all hosts on every tick of the clock.

*/

```
ACTOR clock ::
[
        T : time := 0;

        *[        true → tic &; T := T+1;          ]
]
```

DYNAMIC ALLOCATION, DUAL-PROCESSORS, POINT TO POINT COMMUNICATION



LEGEND

———	Physical Communication Links
- - - - -	Logical Communication Links
○	Ports

1. SYSTEM FUNCTIONALITY

```
/*
Global Definitions - Definition of all variable types, global constants, and global functions.
*/


TYPE coordinate IS INTEGER;
TYPE point IS ARRAY(1..2) OF coordinate;
TYPE elevation IS INTEGER;
TYPE terrain IS ARRAY(*,*) OF elevation;
TYPE distance IS INTEGER;
TYPE relative_distance IS ARRAY(*,*) OF distance;
TYPE road_paths IS ARRAY(*,*) OF BOOLEAN;

x : INTEGER CONSTANT := 1;
y : INTEGER CONSTANT := 2;
e : INTEGER CONSTANT ;

TYPE size IS INTEGER;
TYPE time IS INTEGER;

k : size CONSTANT;

TYPE processing_element IS (1,2);
TYPE host_service IS (allocation;deallocation,computation);
TYPE module_to_processor_mapping IS ARRAY(MODULE) OF PROCESSOR;

initial_hosts : module_to_processor_mapping CONSTANT :=
        (selector : host.1; distance_1 : host.1; distance_2 : host.2;
         minpath : host.2; env1 : host.1; geography : host.2);

COST(s : host_service; q : size) : time ::
[
        [       s = allocation          →       COST := 1       #
                s = deallocation        →       COST := 1       #
                s = computation         →       COST := q       ]
];


PCOST(h1,h2 : PROCESSOR; q : size) : time ::
[
        [       (h1 = host.1 and h2 = host.2    →       PCOST := q      #
                (h1 = host.2 and h2 = host.1    →       PCOST := q      ]
];


COMPLEXITY(o : FUNCTION; q : size) : size ::
[
        [       o = Equ_distance        →       COMPLEXITY := q                 #
                o = Min_distance        →       COMPLEXITY := 5 * q             ]
];
```

```
NEW_HOST(h : PROCESSOR) : PROCESSOR ::
[
        [       h = host.1      →      NEW_HOST := host.2  #
                h = host.2      →      NEW_HOST := host.1  ]
];



Area_spec_ok(e1,e2:point): BOOLEAN ::
[
        Area_spec_ok :=         e1(x) ≤ e2(x)  and  e1(y) ≤ e2(y)
];

End_points_ok(a1,a2,r1,r2:point): BOOLEAN ::
[

        End_points_ok :=        a1(x) ≥ r1(x) and a1(x) ≤ r2(x) and a1(y) ≥ r1(y) and a1(y) ≤ r2(y) and
                                a2(x) ≥ r1(x) and a2(x) ≤ r2(x) and a2(y) ≥ r1(y) and a2(y) ≤ r2(y)
];

Min_distance(D:relative_distance; T:terrain; e:integer): relative_distance ::
[
        /* body of function */
];

Equ_distance(D1,D2 : relative_distance; d : distance) : road_path ::
[
        /* body of function */
];


/*
        Specification of the environment and geography processes - The environment process
models the user interaction, getting the area boundary and road endpoints from user. The
geography is the terrain database from which elevations for a given area can be obtained.
*/


MODULE env1 ::
[
/* body of environment process */
]




MODULE geography ::
[
/* body of geography process */
]
```

```
/*
        The selector module gets the area boundary and road endpoints frome the environment,
and then checks that both endpoints of road are within the area given. If the endpoints are in
area, then the area boundary and road endpoints are sent to distance_1 and distance_2 to
determine the minimum distances to all points in area from both the first and second endpoints.
The area boundary is also sent to minpath, which will determine the points on the road.
*/


MODULE selector ::
[
f0 : MODULE CONSTANT := ~;

a1 : point; dcl.f0 $$ (SIZE(a1));
a2 : point; dcl.f0 $$ (SIZE(a2));
r1 : point; dcl.f0 $$ (SIZE(r1));
r2 : point; dcl.f0 $$ (SIZE(r2));

get.f0.env1 $$ (SIZE(a1)+SIZE(a2)+SIZE(r1)+SIZE(r2)) env1?(a1,a2,r1,r2);

[       Area_spec_ok(a1,a2)              → skip #
        not(Area_spec_ok(a1,a2))         → abort        ];

[       End_points_ok(a1,a2,r1,r2)       → skip #
        not(End_points_ok(a1,a2,r1,r2))  → abort        ];

send.f0.distance_1 $$ (SIZE(a1)+SIZE(a2)+SIZE(r1)+SIZE(r2)) distance_1!(a1,a2,r1,r2);
send.f0.distance_2 $$ (SIZE(a1)+SIZE(a2)+SIZE(r1)+SIZE(r2)) distance_2!(a1,a2,r1,r2);
send.f0.minpath $$ (SIZE(a1)+SIZE(a2)) minpath!(a1,a2);

dlt.f0 $$ (SIZE(a1)); dlt.f0 $$ (SIZE(a2));
dlt.f0 $$ (SIZE(r1)); dlt.f0 $$ (SIZE(r2))
]
```

```
/*
        Each distance module is given the boundary of an area and the endpoints of a road, and
then determines the minimum distance from the first road endpoint to every other point in the
area using the terrain information for the area that it gets from geography.
*/


MODULE distance_1 , distance_2 ::
[
f0 : MODULE CONSTANT := ˜;

a1 : point; dcl.f0 $$ (SIZE(a1));
a2 : point; dcl.f0 $$ (SIZE(a2));
r1 : point; dcl.f0 $$ (SIZE(r1));
r2 : point; dcl.f0 $$ (SIZE(r2));
ok : BOOLEAN; dcl.f0 $$ (SIZE(ok));


get.f0.selector $$ (SIZE(a1)+SIZE(a2)+SIZE(r1)+SIZE(r2)) selector?(a1,a2,r1,r2);

[
        Z : terrain(a1(x)..a2(x),a1(y)..a2(y)); dcl.f0 $$ (SIZE(Z));
        D : relative_distance(a1(x)..a2(x),a1(y)..a2(y)); dcl.f0 $$ (SIZE(D));


        send.f0.geography $$ (SIZE(a1)+SIZE(a2)) geography!(a1,a2);
        get.f0.geography $$ (SIZE(ok)) geography?(ok);

        [       ok      → get.f0.geography $$ (SIZE(Z)) geography?(Z)  #
                not(ok) → abort                                          ];

        D(r1) := 0;
        eval.f0 $$ (Min_distance,SIZE(Z)) D := Min_distance(D,Z,e);

        [       D(r2)≠∞ → send.f0.minpath $$ (SIZE(D)+SIZE(r1)) minpath!(D,r1)     #
                D(r2)=∞ → abort                                                     ];

        dlt.f0 $$ (SIZE(Z)); dlt.f0 $$ (SIZE(D))
];

dlt.f0 $$ (SIZE(a1)); dlt.f0 $$ (SIZE(a2));
dlt.f0 $$ (SIZE(r1)); dlt.f0 $$ (SIZE(r2));
dlt.f0 $$ (SIZE(ok))
]
```

```
/*
        The minpath module gets the minimum distances of points in area from each road
endpoint, and determines the points that are on a road of minimum length between the
endpoints, and sends this list to the environment.
*/


MODULE minpath ::
[
f0 : MODULE CONSTANT := ~;

a1 : point; dcl.f0 $$ (SIZE(a1));
a2 : point; dcl.f0 $$ (SIZE(a2));
r1 : point; dcl.f0 $$ (SIZE(r1));
r2 : point; dcl.f0 $$ (SIZE(r2));

get.f0.selector $$ (SIZE(a1)+SIZE(a2)) selector?(a1,a2);

[
        D1 : relative_distance(a1(x)..a2(x),a1(y)..a2(y)); dcl.f0 $$ (SIZE(D1));
        D2 : relative_distance(a1(x)..a2(x),a1(y)..a2(y)); dcl.f0 $$ (SIZE(D2));

        +[      get.f0.distance_1 $$ (SIZE(D1),SIZE(r1)) distance_1?(D1,r1) → skip  #
                get.f0.distance_2 $$ (SIZE(D2),SIZE(r2)) distance_2?(D2,r2) → skip  ];

        [
                RD : road_paths(a1(x)..a2(x),a1(y)..a2(y)); dcl.f0 $$ (SIZE(RD));
                d : distance; dcl.f0 $$ (SIZE(d));

                d := D1(r2);
                eval.f0 $$ (Equ_distance,SIZE(D1)) RD := Equ_distance(D1,D2,d);
                        send.f0.env1 $$ (SIZE(RD)) env1!RD;

                        dlt.f0 $$ (SIZE(RD)); dlt.f0 $$ (SIZE(d))
                ];

                dlt.f0 $$ (SIZE(D2)); dlt.f0 $$ (SIZE(D1))
        ];

        dlt.f0 $$ (SIZE(a1)); dlt.f0 $$ (SIZE(a2));
        dlt.f0 $$ (SIZE(r1)); dlt.f0 $$ (SIZE(r2))
]
```

2. SYSTEM ARCHITECTURE

```
/*
        The host process models the operations required of our physical hardware. This model is
for memory allocation (total amount, not placement) so the operations modelled are the
allocation of memory and the deallocation (free) of memory.
*/


PROCESSOR host.i" (i : processing_element) ::
[
        M : size := 0;
        n : size;
        p : MODULE;
        t : time;
        h : PROCESSOR;
        h0 : PROCESSOR CONSTANT := ˜ ;

        *[
                allocate.h0.p" @ (n') →
                        [       M+n≤k → M := M+n; success.h0.p @  #
                                M+n>k → skip; failure.h0.p @       ];
                        t := COST(allocation,n); t*(tic &); @@
                                                                          #
                free.h0.p" @ (n') →
                        [       M-n≥0 → M := M−n; success.h0.p @  #
                                M-n<0 → skip; failure.h0.p @       ];
                        t := COST(deallocation,n); t*(tic &); @@
                                                                          #
                compute.h0.p" @ (n') →
                        t := COST(computation,n); t*(tic &); @@
                                                                          #
                pass.h0.h".p".q" @ (n') h!(n) →
                        t := PCOST(h0,h,n); t*(tic &); @@
                                                                          #
                pass.h".h0.p".q" @ (n') h?(n) →
                        t := PCOST(h,h0,n); t*(tic &); @@
                                                                          #
                true → skip; tic &
        ]
]
```

3. SYSTEM SCHEDULER

/*

Each schedule provides the interface between one of the modules and the host process. In this model, this is limited to the allocation and deallocation of memory for variables at the module level, and the allocation and deallocation of temporary storage used in the transfer of data underlying the input/output commands.

*/

```
SCHEDULE server.f* (f : module) ::
[
    mem : size := 0;
    n,req : size;
    o,p : MODULE;
    b : BOOLEAN := true;
    h,h0,h1 : PROCESSOR;

    a_lock.f % (h0'); a_unlock.f % (h0);
    *[
        dcl.f $ (n') a_lock.f % (h0') →
            allocate.h0.f @ (n);
            [   success.h0.f @ → skip;@@                                  #
                failure.h0.f @ →
                    skip; @@;
                    h1 := h0;
                    *[  b →
                            h0 := NEW_HOST(h0);
                            [   h0 = h1 → req := n;         #
                                h0 ≠ h1 → req := mem + n;    ];
                            allocate.h0.f @ (req);
                            [   success.h0.f @ → b:=false;@@ #
                                failure.h0.f @ → skip; @@          ]
                    ];
                    [   h0 ≠ h1 →
                            /* move all the stuff */
                            free.h1.f @ (mem);
                            [   success.h1.f @ → skip;@@;     #
                                failure.h1.f @ → abort        ]     #

                        h0 = h1 → skip                        ]
            ];
            mem := mem + n
            a_unlock.f % (h0); $$
                                                                              #
        dlt.f $ (n') → free.h0.f @ (n);
            [   success.h0.f @ → skip;@@        #
                failure.h0.f @ → abort          ];
            mem := mem - n; $$
                                                                              #
        eval.f $ (o',n') →
            compute.h0.f @ (COMPLEXITY(o,n));
            @@;$$
                                                                      #
        send.f.p" $ (n') c_lock.f.p % (h')→
            allocate.h0.f @ (n);
            [   success.h0.f @ → skip;@@         #
                failure.h0.f @ → abort    ];

            [   h0 = h    →
                    ready.f.p $ (n)                           #
                h0 ≠ h    →
                    pass.h0.h.f.p @ (n); @@;
                    free.h0.f @ (n);
                    [   success.h0.f @ → skip;@@        #
                        failure.h0.f @ → abort          ]     ];
            c_unlock.f.p %; $$
                                                                      #
        get.f.p" $ (n') c_lock.f.p % (h') →.
            [   h0 = h    →
                    ready.p.f $ (n)                           #
                h0 ≠ h    →
                    allocate.h0.f @ (n);
                    [   success.h0.f @ → skip;@@        #
                        failure.h0.f @ → abort          ];
                    pass.h.h0.f.p @ (n); @@;                  ];
            free.h0.f @ (n);
            [   success.h0.f @ → skip;@@         #
                failure.h0.f @ → abort           ];
            c_unlock.f.p %; $$
    ]
]
```

```
/*
        The relocator models the locking and unlocking of the modules required to ensure that a
module does not relocate while a communication is in effect involving it.
*/


SCHEDULE relocator ::
[
        p,q : MODULE;
        h : PROCESSOR;
        avail : ARRAY(MODULE) OF BOOLEAN := true;
        is_a_locked : ARRAY(FUNCTION) OF BOOLEAN := false;
        is_c_locked : ARRAY(FUNCTION) OF BOOLEAN := false;
        host_of : ARRAY(FUNCTION) OF PROCESSOR := initial_hosts;

        *[
                avail(p); a_lock.p" % (host_of(p)); →
                        avail(p) := false;
                        id_a_locked(p) := true
                                                                #
                is_a_locked(p); a_unlock.p" % (h') →
                        avail(p) := true;
                        is_a_locked(p) := false;
                        host_of(p) := h;
                                                                #
                avail(p); avail(q); c_lock.p".q" % (host_of(q)) c_lock.q.p % (host_of(p)) →
                        avail(p) := false;
                        is_c_locked(p) := true;
                        avail(q) := false;
                        is_c_locked(q) := true;
                                                                #
                is_c_locked(q); c_unlock.p".q" % →
                        avail(q) := true;
                        is_c_locked(q) := false;
        ]
]
```

4. PERFORMANCE SPECIFICATION

```
/*
        The actor clock models a global clock that sychronizes with all hosts on every tick of
the clock.
*/


ACTOR clock ::
[
        T : time := 0;

        *[        true → tic &; T := T+1;          ]
]
```

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.

END

12-86

DTIC